IMPROVING DUAL-TREE ALGORITHMS

A Dissertation
Presented to
The Academic Faculty

By

Ryan R. Curtin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2015

IMPROVING DUAL-TREE ALGORITHMS

Approved by:

Dr. David V. Anderson, Committee Chair
Associate Professor, School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Mark A. Clements
Associate Professor, School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Charles L. Isbell, Jr., Advisor
Professor, School of Interactive Computing
Georgia Institute of Technology

Dr. Polo Chau
Assistant Professor, School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard W. Vuduc
Associate Professor, School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: August 18, 2015

This document is dedicated solely to my cats, who do not and will not ever have the capacity to understand even the title of this manuscript, and who, thanks to domestication, are actually entirely incapable of leading any sort of autonomous lifestyle and thus are mortally dependent on my completion of simple maintenance tasks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

THE POINT

This large body of work is entirely centered around *dual-tree algorithms*, a class of algorithm based on spatial indexing structures that often provide large amounts of acceleration for various problems. This work focuses on understanding dual-tree algorithms using a new, tree-independent abstraction, and using this abstraction to develop new algorithms.

Stated more clearly, the thesis of this entire work is that we may improve and expand the class of dual-tree algorithms by focusing on and providing improvements for each of the three independent components of a dual-tree algorithm: the type of space tree, the type of pruning dual-tree traversal, and the problem-specific `BaseCase()` and `Score()` functions. I demonstrate this by expressing many existing dual-tree algorithms in the tree-independent framework, and focusing on improving each of these three pieces.

After historical trivia and an introduction to trees in Chapter 2, Chapter 3 introduces the tree-independent dual-tree algorithm abstraction and notation which will be used throughout the document. Chapter 4 describes **mlpack**, the C++ machine learning library in which most of the advancements in this thesis are implemented. Then, the focus turns to each of the three components of dual-tree algorithms; Chapter 5 focuses on trees, Chapter 6 focuses on pruning dual-tree traversals, and Chapter 7 (a much longer chapter) focuses on new or improved dual-tree algorithms to solve various tasks that are generally related to machine learning. Finally, Chapter 8 concludes the work and sets the stage for all of the potential interesting directions I was not able to consider during my work on this thesis.

An **important note** is that this document is probably not best read cover-to-cover. Only an insane person would do that. Instead, the thesis is more effectively used as a piecemeal reference. As a result, most sections are readable as standalone sections, and where necessary they will reference previous chapters or sections. Therefore, reading about a particular algorithm can generally be done in a depth-first manner, following link chains to learn all

necessary background. Nonetheless, the document is arranged such that it *could* be read serially, in order to appease any readers who are insane.

CHAPTER 2

INTRODUCTION

2.1 An abridged history of statistical computing

Statistical computing as a field first became relevant in the 1920s and 1930s with the widespread adoption of early IBM punched card tabulators [1], after their initial introduction in the late 1800s [2]. These machines made the computation of statistics on non-trivial sets of data feasible (such as the early iris flower dataset by Fisher [3]). Then, the 1940s saw the invention of the transistor [4] and digital computing machines [5], allowing general-purpose computational engines to be easily available from the 1950s onwards [6].

These advancements allowed entirely new types of questions to be answered: data-driven questions. One of the earliest of these to gain popularity was the nearest-neighbor distance [7, 8, 9], which led to the well-known nearest neighbor rule for classification [10]. Numerous other data-driven algorithms for various tasks appeared: neural networks [11], minimum spanning trees [12], the fast Fourier transform [13], maximum-likelihood estimation [14], density estimation [15], sorting [16], matrix decompositions [17], and an enormous host of algorithms for countless tasks.

However, the size of the problems continued to grow. The advent of the microprocessor in 1971 [18], the introduction of the "1977 Trinity" [19]—the Apple II, the Commodore PET, and the TRS-80—and the fulfillment of Moore's law [20] meant that an entirely new generation of programmers and scientists could apply the algorithms of the 1960s and 1970s to continually larger and more difficult problems as computational barriers were demolished.

The trends of increasing computational power leading to increasing dataset size in an interesting positive feedback loop are still in effect today. Virtually every presentation and publication in the field of machine learning, data mining, and statiatical analysis has the same introduction depicting the "data deluge" as a giant computational problem that is

increasingly insurmountable—except with the methods described in that particular publication, of course. Even the popular media has latched onto this phenomenon, with numerous articles devoted to "big data" [21, 22, 23].

Still, one thing that redundant presentation introductions and media outlets alike all have correct is that computational advances are continually pushing the bounds of dataset sizes upwards. This highlights the ever-increasing importance of algorithms that scale well with dataset size, which justifies the study and development of scalable algorithms—and that is the focus of this thesis.

2.2 A less abridged history of the development of trees

When considering large datasets, there are two commonly-employed general approaches: sampling and trees[1]. The sampling school of thought states that not all of the data is necessary: only some small amount of the data is necessary to obtain an approximate solution. This approach is heavily used in the kernel methods subgenre of the machine learning community [24, 25]. The tree-based school of thought states that a dataset may be represented hierarchically: we may select a few points that represent the data at a very high level, then some points that represent the data at a medium level, then many points that reflect the data at a low level, then finally the data itself at the lowest level. This thesis is concerned solely with understanding and improving the second strategy; therefore, an in-depth discussion of sampling approaches is not found here. That may be found elsewhere [26, 27]. In this section, we discuss the history of tree-based algorithms with an eye towards the generalized tree-based algorithm framework that this thesis is largely based on [28].

In the previous section, the nearest-neighbor rule for classification was briefly mentioned [10]. This task is our jumping-off point for trees, so let us consider it formally[2]. We are given some *reference dataset* S_r full of *reference points* $p_r \in S_r$; each reference point

[1] Surely I have not considered every possible approach, but these two are quite standard. It is also worth noting that these approaches are not exclusive: for instance, one may build a tree on sampled data.

[2] This statement of the problem is not true to the original notation. But it is consistent with the rest of the document.

p_r is associated with some *class* c_r, We are also given some *query point* p_q. Our task is to predict c_q.

The nearest neighbor rule for classification states that $c_q = c_{nn}$, where c_{nn} corresponds to p_{nn}, which is the nearest neighbor of p_q in S_r:

$$p_{nn} = \underset{p_r \in S_r}{\operatorname{argmin}}\, d(p_q, p_r) \tag{1}$$

for some distance metric $d(\cdot, \cdot)$. At first glance—and in the first implementation—p_{nn} is simply calculated by iterating over all points in S_r and saving the best result. If the size of S_r is N, this takes $O(N)$ time per query point p_q. While modern computing equipment runs this algorithm in reasonable time for N in the hundreds of thousands, larger datasets present severe computational challenges, and if there exists a sizeable query set S_q instead of just a single query point p_q, the scaling issues are even more severe.

In 1974, Finkel and Bentley [29] proposed a multidimensional binary space tree called a 'quadtree', in the context of information retrieval systems and databases. The quadtree is a hierarchical indexing structure that requires the data S_r to lie in two dimensions. The top level (the *root*) of a quadtree corresponds to a square which encompasses the entire dataset S_r. The root has up to four children, each corresponding to the four half-size squares that fit in the square represented by the root. Each of these children is split in the same way recursively until the node contains at most some specified number of points (call this the *leaf size*), and these leaf nodes contain each of the points in S_r which lie in the square represented by the leaf node.

A quadtree may be best explained visually; to that end, Figure 1 displays the abstract

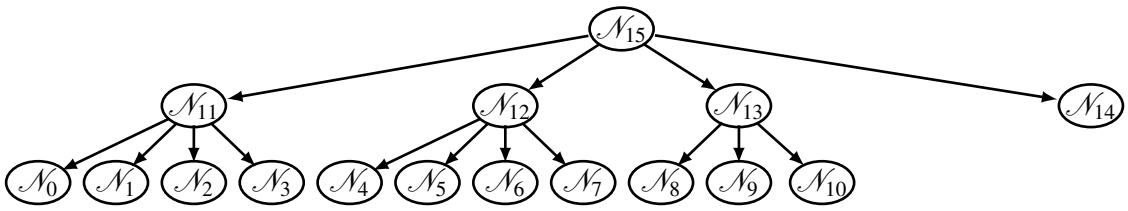
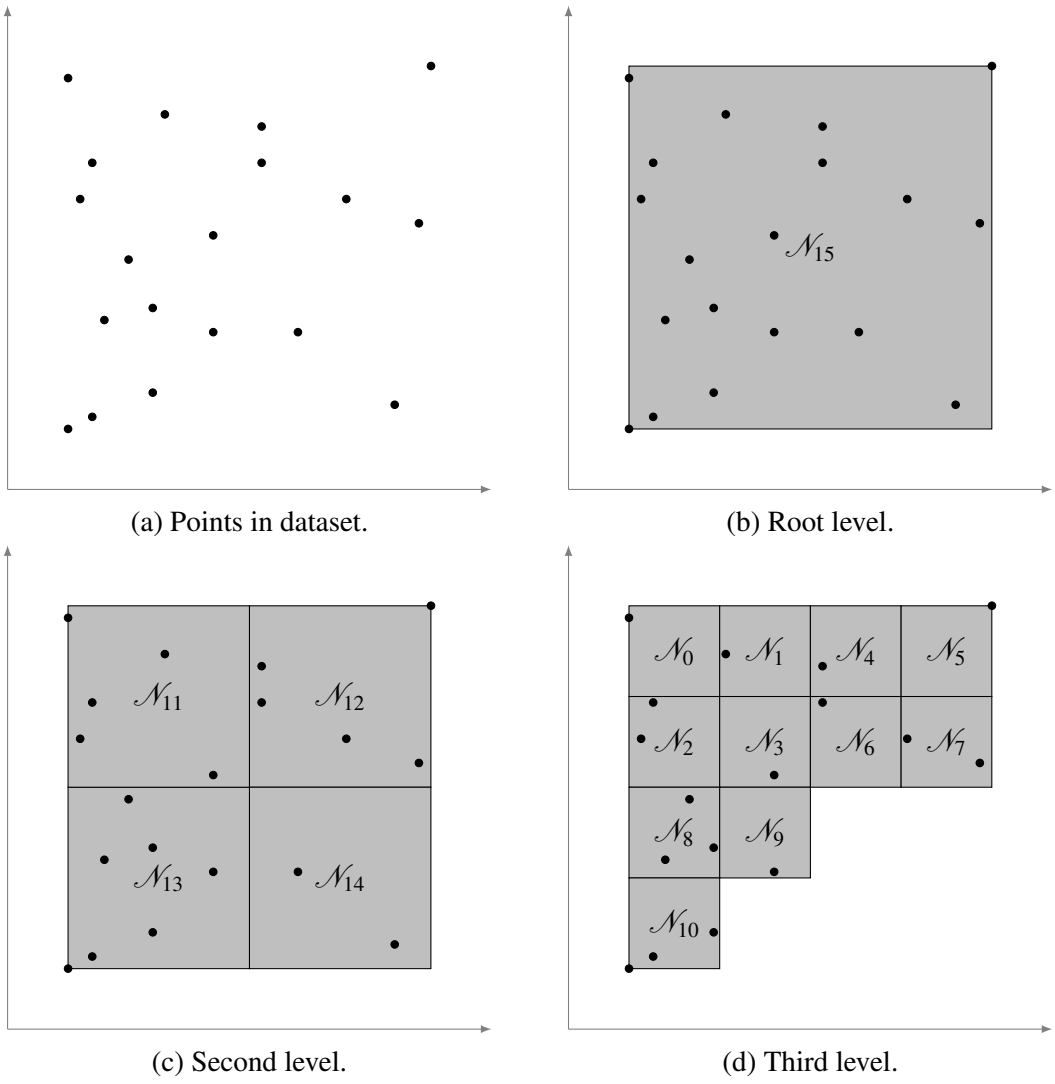


Figure 1: Abstract representation of an example quadtree.



(a) Points in dataset.

(b) Root level.

(c) Second level.

(d) Third level.

Figure 2: Geometric representation of the same example quadtree.

representation of an example quadtree, and Figure 2 display the representation of three levels in $\mathcal{R}^2$ of the same example quadtree. The points (Figure 2a) all lie in the square represented by the root node $\mathcal{N}_{15}$ (Figure 2b). Then, $\mathcal{N}_{15}$ is split into four children: $\mathcal{N}_{11}$, $\mathcal{N}_{12}$, $\mathcal{N}_{13}$, and $\mathcal{N}_{14}$; each of these correspond to a square with half the side length as $\mathcal{N}_{15}$. Then, if the node contains more than three points[3], it is split again; yielding the lowest level of nodes, shown in Figure 2d (note that in this last figure, the points held in $\mathcal{N}_{14}$ are

[3]Three, the leaf size here, is selected arbitrarily. Different choices are of course possible.

not shown). The node which would be the fourth child of $\mathcal{N}_{13}$ does not hold any points—therefore, it does not need to be a part of the tree. Quadtrees built on larger or different datasets have the same type of structure.

Finding the nearest neighbor of S_r using a quadtree amounts to a depth-first search with backtracking where at any time the nearest neighbor candidate $\hat{p}_{nn}$ is cached[4]. Then, if the minimum distance between p_q and the square represented by any node $\mathcal{N}_i$ is greater than $d(p_q, \hat{p}_{nn})$, then no descendant point of $\mathcal{N}_i$ can possibly hold the nearest neighbor of p_q, and the search does not need to recurse into any children of $\mathcal{N}_i$. In this way, the search for p_{nn} is greatly accelerated and takes far less than $O(N)$ time for a single query point.

The quadtree was later generalized, in 1980, to the octree [30], which works in three dimensions instead of two. Each node in an octree has eight children, instead of four. Further generalization to arbitrary dimensions is possible, but has a clear problem: in d dimensions, each node will have up to 2^d children.

As an effort to work around this unfavorable exponential dependence on dimension, in 1975, Bentley [31] proposed the *kd*-tree: this structure is far more effective in high-dimensional settings. The idea is simple and related to the quadtree: given a dataset $S_r \in \mathcal{R}^d$, we build a hierarchical structure where each node in the hierarchy corresponds to some region of $\mathcal{R}^d$. In a *kd*-tree, each node $\mathcal{N}_i$ may have a *left child* and a *right child*. Splitting the region represented by a node $\mathcal{N}_i$ into the region represented by its left child $\mathcal{N}_l$ and right child $\mathcal{N}_r$ may be done many ways, but it always involves an axis-aligned split. This means choosing a dimension to split on (often dimensions are chosen sequentially or as the dimension with maximum data variance) and choosing a value to split on (often the median or mean of the data in the chosen split dimension). The left child will correspond to the region required to encompass the data with value less than the split value in the split dimension, and the right child will correspond to the region required to encompass the data with value greater than or equal to the split value in the split dimension.

[4] I am hand-waving here, but a detailed algorithm will be given shortly.

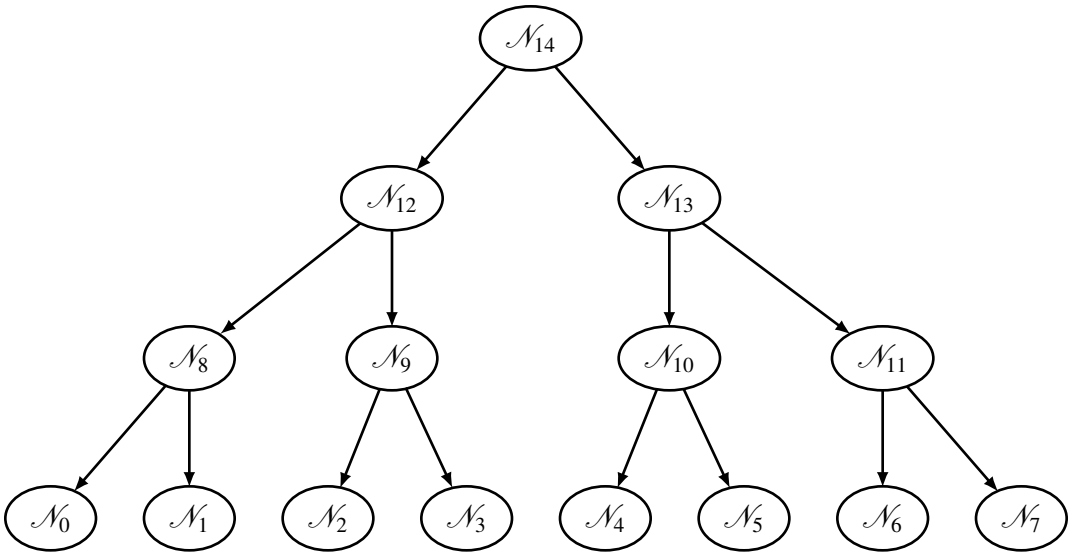
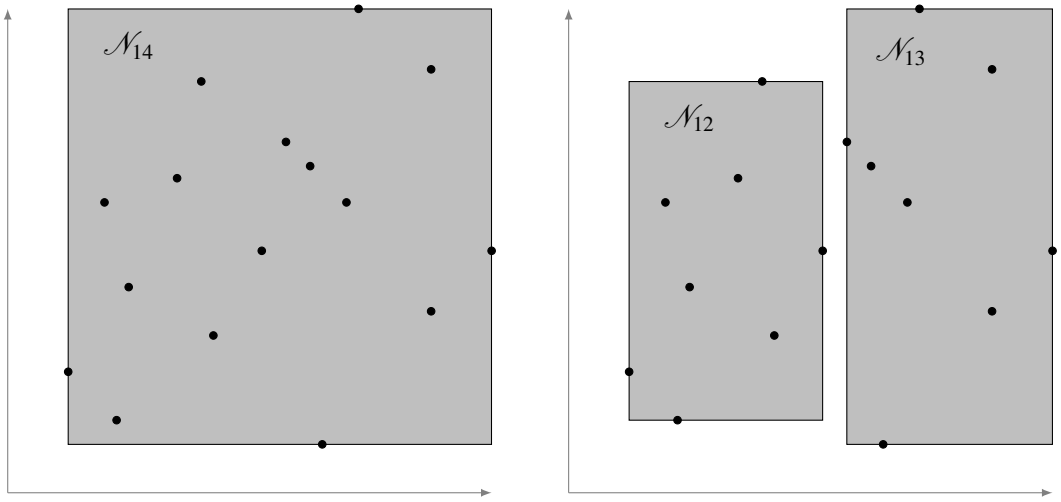


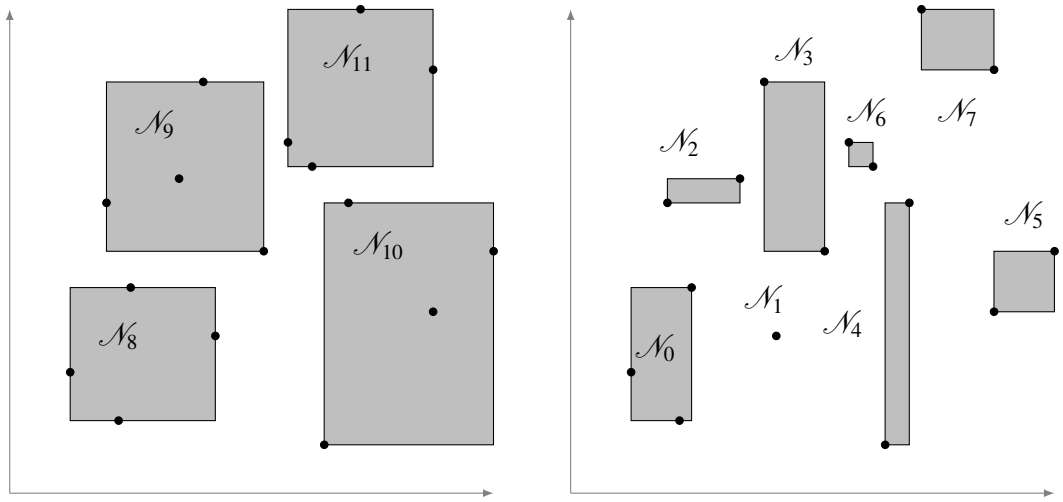
Figure 3: Abstract representation of an example *kd*-tree.

The *kd*-tree is visually described in Figure 3, as an abstract tree, and in Figures 4 and 5, in $\mathcal{R}^2$. Similar to the quadtree, the root node encompasses all of the points. The first split dimension is along the horizontal axis; points with horizontal axis value less than the split value are grouped into the left node ($\mathcal{N}_{12}$) and points with horizontal axis value greater than the split value are grouped into the right node ($\mathcal{N}_{13}$). Note that the bounding rectangles for the child nodes are the smallest bounding rectangles that contain all of the points, so they



(a) Top level. (b) Second level.

Figure 4: Geometric representation of the same example *kd*-tree (top two levels).



(a) Third level.

(b) Lowest level (leaves).

Figure 5: Geometric representation of the same example *kd*-tree (bottom two levels).

may be smaller than the split rectangles of the root. For instance, node $\mathcal{N}_1$ (Figure 5b) contains only one point and is thus a rectangle with zero area (i.e. just a point).

The use of a *kd*-tree to perform nearest neighbor search is similar to the use of a quadtree. An algorithm is given in Algorithm 1. This algorithm is a depth-first traversal that begins at the root of the *kd*-tree node, and caches a current nearest neighbor candidate

Algorithm 1 `kd_nn()`: nearest neighbor search using a *kd*-tree.

Input: query point p_q, reference *kd*-tree node $\mathcal{N}_r$

{Prune if possible.}
if $d_{\min}(p_q, \mathcal{N}_r) > d(p_q, p_r^*)$ **then**
  return

{Perform base cases.}
if $\mathcal{N}_r$ is a leaf **then**
  for all points p_r held in $\mathcal{N}_r$ **do**
    if $d(p_q, p_r) < d(p_q, p_r^*)$ **then**
      $p_r^* \leftarrow p_r$

{Recurse.}
if $\mathcal{N}_r$ is a leaf **then**
  return
else
  `kd_nn(`p_q, left child of $\mathcal{N}_r$`)`
  `kd_nn(`p_q, right child of $\mathcal{N}_r$`)`

p_r^*. The recursion traverses the tree, attempting to prune based on geometric reasoning: if the minimum distance between a node $\mathscr{N}_r$ and p_q, denoted $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$, is greater than the distance between p_q and its current nearest neighbor candidate, then we can reason that no point held in any descendant node of $\mathscr{N}_r$ can possibly be closer to p_q than p_r^*, and thus we can prune that node. This check is performed in line 2.

If a node is not pruned, and it is a leaf node (that is, if it has no children), then each point in the node is compared with p_q in an attempt to improve the nearest neighbor candidate p_r^* (lines 5–7).

At the end of the traversal, p_r^* will contain the nearest neighbor of p_q that is in S_r. This is easy to prove: one may first show that if no pruning occurs, p_r^* will be correct because every point in S_r will be searched. Then, one must simply show that the pruning rule never prunes away any point which could be the nearest neighbor of p_q, and correctness is thus proven. Despite the fact that the search performs backtracking, an expected runtime of $O(\log N)$ can be shown [31, 32].

Extension of Algorithm 1 to the quadtree case simply involves modifying the recursion to visit each of the quadtree's four children (as opposed to the *kd*-tree's two). Further extensions and improvements of the algorithm as given do exist; Algorithm 1 differs wildly from Friedman's implementation [32], but the goal here is to present the algorithm as simply as possible for discussion purposes.

2.3 The explosion of single-tree algorithms

The nearest neighbor search algorithm for *kd*-trees given in the previous section (and its quadtree extension) may be referred to as *single-tree algorithms*[5], as they construct a single tree on the reference dataset and traverse the tree in order to solve the problem. But nearly simultaneous to Bentley's developments was Fukunaga's exploration of a more generalized

[5]The term *single-tree algorithm* is probably specific to myself and other members of Alex Gray's lab; these algorithms may also be known in other circles as *tree-based algorithms*, *tree algorithms*, and/or *branch-and-bound algorithms*. I will, however, use my term, in order to differentiate between single-tree and dual-tree algorithms.

branch-and-bound algorithm for finding k-nearest neighbors [33], except for that the tree structure proposed differed significantly: the children are created by running the k-means clustering algorithm. This structure may be referred to as the *k-means tree*, and was also applied to clustering [34] and feature selection [35].

These handful of algorithms spurred the development of numerous algorithms; a partial list (with quick descriptions of the problem being solved) is given below.

- **minimum spanning tree calculation** [36]: given a dataset S_r, calculate the spanning tree with minimum total edge distance.
- **range search** [37]: given a query point p_q, a range $[l, u]$, and a reference dataset S_r, find every point in S_r with distance to p_q in the range $[l, u]$.
- **approximate nearest neighbor search** [38]: given a query point p_q and a reference set S_r, find the approximate nearest neighbor of p_q; there are a multitude of approximation schemes, but the most common is probably relative-value approximation, where the nearest neighbor returned must have distance no more than $(1 + \epsilon)$ times the true nearest neighbor distance.
- **k-means clustering** [39]: given a dataset S_r and a number k, find k clusters by minimizing the within-cluster sum of squared distances.
- **training Gaussian mixture models** [40]: given a dataset S_r and a number k, fit k Gaussians to S_r.
- **ray tracing** [41]: given a collection of objects, light sources, and a camera location, trace the path of light through the space and account for its interaction with the objects; a common application is the generation of realistic images.
- **solving the gravitational n-body problem** [42]: given a set of particles S_r, calculate the gravitational force on a query particle p_q exerted by every particle in S_r (often, instead of a single query particle p_q, results are required for every particle in S_r).

- **Gaussian process regression** [43]: a flexible regression technique that interpolates the observations and allows confidence intervals for predictions.

- **kernel regression** [44]: a regression technique where given a kernel function and a dataset S_r, the prediction at a particular query point p_q is a weighted combination of the predictions for points in S_r.

- **maximum inner product search** [45]: given a dataset S_r and a query point p_q, find the point in S_r with maximum inner product with p_q.

- **max-kernel search** [46]: a generalization of maximum inner product search; given a kernel function $\mathcal{K}(\cdot, \cdot)$, a dataset S_r, and a query point p_q, find the point in S_r with maximum kernel function evaluation with p_q.

These are only a few of the countless existing single-tree algorithms. These algorithms, as proposed, often used various different types of tree structures; sometimes, tree structures were proposed independently. In general, each single-tree algorithm above is adaptable to different types of trees, but at that time there was no formalized notion of tree or single-tree algorithm and thus each adaptation required special handling and care.

2.4 A smorgasboard of trees

We have only discussed the *kd*-tree and quadtree in any detail, but it should be clear that the design space for trees is enormous. Roughly speaking, a tree is any sort of hierarchical indexing structure, and there is huge flexibilty in how the children of a node are selected and created. Some trees are known to work better for some problems than others, and some trees work better for certain types of data than others. This no-clear-winner conundrum led to incomprehensible numbers of different techniques for tree-building and a mystifying assortment of different trees suited to different tasks. A partial list of many tree types is given below; note that it is (very) incomplete! An important observation is how different the structures of these trees are.

- **quadtree** [29] (1974): splits 2-d space into four nodes.
- ***kd*-tree** [31] (1975): splits data according to axis-aligned splits; has two children.
- ***k*-means tree** [33] (1975): data is split using k-means clustering, recursively.
- **octrees** [30] (1980): a generalization of quadtrees to 3 dimensions, with 8 children for each node.
- **R trees** [47] (1984): a height-balanced tree similar to the B tree, optimized for dynamic insertions and removals.
- **ball trees** [48] (1989 or earlier): each node represents a ball in the input space; nodes may end up overlapping depending on the construction technique used.
- **R* trees** [49] (1990): a modified R tree which has better optimization of node area during point insertion.
- **vantage-point trees** [50] (1993) / metric trees [51] (1991): each (ball-shaped) node has two children: one child corresponds to those points near the center of the ball, and the other corresponds to the points far away from the center.
- **Hilbert R trees** [52] (1994): an improvement on older R tree variants, which forces a linear ordering on the nodes to improve search time.
- **TV trees** [53] (1994): indexes high-dimensional data by ignoring all but a few features.
- **X trees** [54] (1996): an optimization of R trees to high-dimensional settings that uses 'supernodes'.
- **principal axis trees** [55] (2001): each node splits its children along the principal axis of its subset of the data.

- **spill trees** [56] (2004): a modified *kd*-tree which allows nodes to overlap and points to be held in multiple leaves; developed for approximate nearest neighbor search.
- **cover trees** [57] (2006): a fascinatingly complex tree structure specialized for proving worst-case runtime bounds with respect to properties of the dataset.
- **cosine trees** [58] (2008): the cosine similarity is used to split points into "similar" and "dissimilar" points.
- **max-margin trees** [59] (2012): a binary tree where each node split enforces a robust separation of the data, in order to minimize the number of nodes searched to find the true nearest neighbor.
- **cone trees** [45] (2012): each node corresponds to those points which lie in a cone around a vector; this is specialized for maximum inner-product search.

In general, every tree type listed above can be used to solve each of the problems listed in the previous section, but often some amount of adaptation is necessary. For instance, the nearest neighbor search algorithm for the cover tree [57] differs significantly from the nearest neighbor search algorithm as given in Algorithm 1.

2.5 The fast multipole method and query amortization

The intuition that eventually led to this thesis was developed at approximately the same time as the author in 1987; it is a well-known algorithm called the 'fast multipole method' (or more colloquially, 'FMM') for the calculation of pairwise interactions of particles in particle simulations, due to Greengard and Rokhlin [60][6]. To demonstrate the advancement of Greengard and Rokhlin's algorithm, let us first consider an incrementally older single-tree algorithm by Barnes and Hut [42] that solves the same problem.

[6]Was the fast multipole method the first algorithm to amortize work over query points? Maybe not—but it is certainly the earliest well-known work that could be considered a dual-tree algorithm, and the eventual development of dual-tree algorithms traces its origins directly to Greengard and Rokhlin's algorithm [61].

Algorithm 2 Barnes-Hut force calculation for quadtrees: `bh_quadtree()`.

Input: quadtree node $\mathscr{N}_i$, particle p_q, approximation parameter θ, force estimate $\hat{F}_q$.

{Attempt to approximate and prune, if possible.}

if $\left(\text{sidelength}(\mathscr{N}_i) \;/\; d(p_q, \text{com}(\mathscr{N}_i))\right) < \theta$ **then**

  $\hat{F}_q \leftarrow \hat{F}_q + \left(G\frac{m_q \,\text{tw}(\mathscr{N}_i)(p_q - \text{com}(\mathscr{N}_i))}{\|p_q - \text{com}(\mathscr{N}_i)\|^3}\right)$

  return

{If we are a leaf, add the exact contributions of the points we hold.}

if $\mathscr{N}_i$ is a leaf **then**

  for all points p_i held in $\mathscr{N}_i$ **do**

    $\hat{F}_q \leftarrow \hat{F}_q + \left(G\frac{m_q m_i (p_q - p_i)}{|p_q - p_i|^3}\right)$

{Recurse into children.}

for all children $\mathscr{N}_c$ of $\mathscr{N}_i$ **do**

  Call `bh_quadtree()` with $\mathscr{N}_c$ and p_q.

Our problem is to solve the gravitational N-body problem: we are given some set S_r of points which generally lie in either $\mathcal{R}^2$ or $\mathcal{R}^3$ at time t. Our task is to compute (approximately) the position of each of the particles in S_r at time $t + \epsilon$ for some given time step ϵ. Given that each point $p_i \in S_r$ has mass m_i, the core of this task may be expressed as computing the force F_i on each point $p_i \in S_r$:

$$F_i = -\sum_{p_j \neq p_i, p_j \in S_r} G\frac{m_i m_j (p_i - p_j)}{\|p_i - p_j\|^3}. \tag{2}$$

Notice that as $\|p_i - p_j\|$ becomes large, the force interaction between p_i and p_j becomes very small. Therefore, if we allow some amount of approximation, we may ignore or approximate those calculations where p_i and p_j are very far apart. This reasoning is similar enough to the type of reasoning used to prune away nearest neighbor search that it is not too hard to see the outlines of a single-tree algorithm. In Algorithm 2, we show psuedocode for Barnes and Hut's $O(N \log N)$ algorithm for solving this problem, specialized to $\mathcal{R}^2$ and assuming a quadtree $\mathscr{T}$ has been built on S_r. When the quadtree is built, the center-of-mass of each node, denoted com(·), is calculated and cached, as well as the total weight, denoted tw(·).

The exposition here differs significantly from—and is far more readable than—the arcane SCHEME code given in the original paper [42]. Given some query point $p_q \in S_r$, some approximation parameter $\theta \sim 1$, and starting at the root of $\mathscr{T}$, the algorithm calculates $\hat{F}_q$, an approximation to F_q. As θ is increased, the approximation at line 3 happens at higher and higher levels of the recursion.

A rough expected-time analysis by Barnes and Hut shows that the force calculation for a single point takes $O(\log N)$ time assuming a homogenous mass distribution over S_r. This means the time to calculate the force for every particle—and thus to perform the N-body simulation for a single time-step—takes $O(N \log N)$ time.

An important note about Barnes and Hut's algorithm is that we must iterate over every point in S_r for a single time step. However, this is not strictly necessary, and Greengard and Rokhlin's FMM (which can be seen, with some mental gymnastics, as the 'first dual-tree algorithm') shares work across points in S_r.

The details of the FMM are quite complex and for the sake of this discussion unnecessary, but the entire algorithm depends on the *multipole expansion*, which is, roughly, an approximation of Equation 2 as an order-p polynomial (the choice of p controls the approximation level, unlike θ in Barnes and Hut's algorithm). The key observation is that given some multipole expansion about some point p_i, we may *translate* the expansion to be about some other point p_j. Thus, we can form multipole expansions about the centers of the various nodes in our tree, and translate them to the centroids of other nodes.

The algorithm itself, then, consists of two passes over a tree: an *upward* pass, where multipole expansions are computed about nodes in the tree, and a *downward* pass, where these pre-computed multipole expansions are translated to other nodes in the tree and expanded to provide a force estimate for each point in S_r. This only requires two passes over the tree, and Greengard and Rokhlin claimed worst-case $O(N)$ running time (for $|S_r| = N$). This claim was later contested by Aluru [62]; however, the FMM is known to scale linearly

in practice, instead of as $O(N \log N)$, as in Barnes and Hut's single-tree algorithm. Empirical results provided by both papers established each algorithm as an effective and efficient alternative to brute-force calculation[7].

The importance of the fast multipole method to the computational physics community and related communities cannot be understated; it has spawned more descendant literature than is worth citing here and the careers of numerous researchers have focused entirely on applications of and improvements to the fast multipole method.

What we can take out of the FMM from this quick discussion, instead of the details of the algorithm, is the intuition that is used. Through the use of the multipole expansion, we exploit the fact that two nearby points have very similar interactions with faraway points. That is, the work to compute force interactions for nearby points is amortized across those points and is not unnecessarily duplicated, as in Barnes and Hut's algorithm.

2.6 Redirection to statistics and dual-tree algorithms

Finally, we can return to the problem of nearest neighbor search. Suppose now that instead of a single query point p_q, we have an entire *query set* S_q, and we must find the nearest neighbor of every $p_q \in S_q$ in the *reference set* S_r. This is sometimes called the *batch nearest neighbor search* problem. The work of Gray and Moore in 2001 adapted the fast multipole method to the problem of nearest neighbor search (and a few other problems) in order to obtain *dual-tree algorithms* [61].

In short, the idea is this: instead of traversing the tree built on S_r for each query point $p_q \in S_q$, we will also build a *query tree* on S_q, and traverse both trees (the query tree and reference tree) simultaneously, in order to obtain results for all points in S_q during a single

[7]Both of these results were obtained on VAX machines; it seems as though Greengard and Rokhlin had access to nicer equipment, having run their simulations on the then-recent VAX 8600, whereas Barnes and Hut's results were on the older VAX 11/780. Sadly, the VAX architecture is all but dead now except in the hands of collectors[8], after the implosion of DEC in the late 1980s, precipitated in part by the company's failure to recognize the importance of the PC market [63], which led to the acquisition of DEC by Compaq, which later become a part of HP, which has had more than its fair share of issues over the years.

[8]The author is an unsuccessful DEC collector.

Algorithm 3 `dual_kd_nn()`: Nearest-neighbor search using two *kd*-trees.

Input: query tree node $\mathscr{N}_q$, reference tree node $\mathscr{N}_r$

{Calculate bound, and prune if possible.}
{$\mathscr{D}^p(\mathscr{N}_q)$ represents the set of descendant points of the query node $\mathscr{N}_q$.}
$b \leftarrow \max_{p \in \mathscr{D}^p(\mathscr{N}_q)} D_{p_q}$
if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > b$ **then**
  return

{Perform base cases.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    if $d(p_q, p_r) < D_{p_q}$ **then**
      $N_{p_q} \leftarrow p_r$
      $D_{p_q} \leftarrow d(p_q, p_r)$

{Dual-tree recursion.}
if $\mathscr{N}_q$ is a leaf and $\mathscr{N}_r$ is a leaf **then**
  return
else if $\mathscr{N}_q$ is a leaf **then**
  `dual_kd_nn`($\mathscr{N}_q$, left child of $\mathscr{N}_r$)
  `dual_kd_nn`($\mathscr{N}_q$, right child of $\mathscr{N}_r$)
else if $\mathscr{N}_r$ is a leaf **then**
  `dual_kd_nn`(left child of $\mathscr{N}_q$, $\mathscr{N}_r$)
  `dual_kd_nn`(right child of $\mathscr{N}_q$, $\mathscr{N}_r$)
else
  `dual_kd_nn`(left child of $\mathscr{N}_q$, left child of $\mathscr{N}_r$)
  `dual_kd_nn`(left child of $\mathscr{N}_q$, right child of $\mathscr{N}_r$)
  `dual_kd_nn`(right child of $\mathscr{N}_q$, left child of $\mathscr{N}_r$)
  `dual_kd_nn`(right child of $\mathscr{N}_q$, right child of $\mathscr{N}_r$)

traversal.

To demonstrate this, let us adapt Algorithm 1 to a dual-tree algorithm. This new dual-tree algorithm, Algorithm 3, uses *kd*-trees and is similar to the dual-tree algorithm given by Gray and Moore [61] to calculate the two-point correlation. Before running the algorithm, we build a query tree $\mathscr{T}_q$ on the query set S_q, a reference tree $\mathscr{T}_r$ on the reference set S_r, and initialize two auxiliary arrays: N, where N_{p_q} contains the current nearest neighbor candidate of a query point p_q, and D, where D_{p_q}, which contains the distance between p_q and its current nearest neighbor candidate N_{p_q}. Each element of D should be initialized to ∞. When this initialization step is complete, we call Algorithm 3 with the root of the query

tree and the root of the reference tree as arguments.

This algorithm bears many similarities to the single-tree algorithm. In the single-tree algorithm, the pruning rule only needs to compare against the current nearest neighbor candidate distance for the single query point p_q; but in the dual-tree algorithm, we must consider the nearest neighbor candidate distances for every descendant point of the query node. It is possible to cache the calculation on line 2 during the traversal; for simplicity of exposition, those details are omitted[9]. The base case `for` loop is also similar—but in the dual-tree algorithm, we now have multiple query points to consider, so we have a double `for` loop. As in the single-tree algorithm, no base cases are performed unless $\mathscr{N}_q$ and $\mathscr{N}_r$ are both leaf nodes. Lastly, the recursion is slightly more complex: it must recurse into both the query and the reference node simultaneously, if they are not leaves.

Similar to the single-tree algorithm, a correctness proof of Algorithm 3 is not very difficult. The first step is to show that if nothing is pruned, the correct results are obtained for each $p_q \in S_q$. Then, the second step is to show that no combinations of query and reference nodes are pruned when they should not be. Some of my previous work [28] contains a correctness proof, which will be restated later in this document in a more comprehensive and generalized form.

Like the fast multipole method, the key here is that work is amortized across queries; we do not need to perform separate searches for each query point. In practice, for $|S_q| \sim |S_r| \sim O(N)$, this dual-tree nearest neighbor search algorithm scales linearly and provides massive speedup over other approaches (in low-to-medium dimensions).

It turns out that there are many problems that dual-tree algorithms may solve; often, to develop these algorithms, the single-tree approaches referenced in previous sections may be adapted (with some effort). Below is a nearly-comprehensive list of problems that have been solved with dual-tree algorithms, including my own contributions.

[9] One may refer to the works of Gray [64, 65] for examples of how bounding information can be efficiently cached during the dual depth-first traversal of *kd*-trees.

- **Nearest neighbor search** [61, 57, 28, 66]: given a query set S_q and reference set S_r, find the nearest neighbor of each point in S_q in S_r.

- **Approximate nearest neighbor search** [67, 59]: the same as the above problem, but allowing approximate neighbors to be returned, according to various approximation schemes.

- **Minimum spanning tree calculation** [68]: given a dataset S_r, calculate the spanning tree with minimum edge weight.

- **Kernel density estimation** [65, 64, 69]: given a kernel function, a reference set S_r, and a query set S_q, compute the approximate kernel density estimate at each point in the query set.

- **Conditional kernel density estimation** [70]: an extension of the kernel density estimation problem where conditional density estimates are required (i.e. $f(x|y)$ instead of $f(x, y)$).

- **Approximate matrix multiplication** [71]: given two matrices S_q and S_r, approximately compute $S_q \cdot S_r$.

- **Mean shift clustering** [72]: using a kernel function, locate the density maxima of a dataset S_r, and use these maxima to define a clustering of the dataset.

- **Gaussian summations** [73, 74]: given a query set S_q and a reference set S_r, compute the sum of Gaussian kernel interactions with every reference point, for every query point.

- **Generalized kernel summations** [75, 76]: similar to the above problem, but generalized to any type of shift-invariant kernel.

- ***n*-point correlation function estimation** [77, 78]: a common technique used in astronomy to obtain a statistic useful for describing structure formation models.

- **Maximum inner product search** [45]: given a query set S_q and a reference set S_r, compute the reference point with maximum inner product to each query point.
- **Max-kernel search** [46, 79]: a generalization of maximum inner product search; given a query set S_q, a reference set S_r, and a kernel $\mathcal{K}(\cdot, \cdot)$, compute the reference point with maximum kernel evaluation to each query point.
- **k-means clustering** [80]: given a dataset S_r and a number k, find k clusters by minimizing the within-cluster sum of squared distances.
- **Particle smoothing** [81]: given a sequence of observations S_r, compute a smoothed estimate of those observations.
- **T-SNE (embedding)** [82]: given a potentially high-dimensional dataset S_r, embed S_r into a few dimensions in a way that effectively captures the distribution of the data at both large and small scales.
- **Approximate matrix-vector multiplication** [83]: given a vector, quickly multiply it against a data matrix.
- **Mode seeking** [84]: a generalization of the mean-shift algorithm; given a dataset S_r, find the modes of the distribution of points, usually for the task of clustering.
- **Transition matrix approximation** [85]: given a data graph, approximate the transition matrix for random walks on that graph.

At this point, we have seen three lists, containing numerous types of trees, a veritable plethora of single-tree algorithms, and a great deal of dual-tree algorithms. Clearly tree-based approaches are useful, given that there is no dearth of literature concerning them. But there is one very important fact that this quick overview has downplayed: each of these types of trees and each of these algorithms are significantly different; there is no coherence or unification. It is like the worst of urban sprawl in large American cities:

each algorithm or tree type is its own closed-off neighborhood, lacking connections to other algorithms or tree types. There is no master plan or big picture; these algorithms are developed haphazardly for individual tasks. Although usually the papers in which these algorithms are developed do cite other relevant works, the terminology and notation is not standardized and thus translating the core ideas of one algorithm to another can often be complex, unwieldy, and time-consuming.

To me, this is a significant problem, and the next chapter attempts a solution: a unified abstraction to tie together all types of trees, all types of single-tree algorithms, and all types of dual-tree algorithms, in order to organize the landscape of tree-based algorithms, allow easy knowledge transfer between algorithms, and simplify the development of new tree types, single-tree algorithms, and dual-tree algorithms.

CHAPTER 3

TREE-INDEPENDENT DUAL-TREE ALGORITHMS

3.1 A bibliographical note

The work in this chapter is an extended and somewhat rewritten version of the paper "Tree-independent dual-tree algorithms", by myself and numerous helpful coauthors, which was presented at ICML 2013 [28]. This work formalizes a lot of intuitive but informal notions that had been floating around the community and generalizes the entire classes of single-tree and dual-tree algorithms. The abstractions, definitions, and notation introduced in this chapter are used throughout the rest of the document.

3.2 The goal: unification of dual-tree algorithms

The previous chapter outlined the history of dual-tree algorithms, concluding by observing that the landscape of dual-tree algorithms (at least as of 2013) was not unified, was confusing, and it took a great deal of effort to develop new algorithms. In practice, a researcher may have had to implement entirely separate algorithms to solve the same problems with different trees, which is time-consuming and clearly suboptimal. Worse yet, parallel dual-tree algorithms are difficult to develop (for an example see Lee's work [76], for which the associated code took many months to develop) and appear far more complex than serial implementations; yet, both are solving the same problem.

This chapter introduces a formalizing abstraction for dual-tree algorithms, allowing us to address the issues above with the following tools:

- Formalized definitions of **space trees** and **traversals**.
- A **representation** of dual-tree algorithms as **three separate components**: a space tree, a traversal, and problem-specific rules: a point-to-point base case and a pruning rule.

- A **meta-algorithm** that produces dual-tree algorithms, given those four separate components.

This representation of dual-tree algorithms also has favorable implications for theoretical work as well as implementation.

3.3 Space trees

The first thing to do is formalize the notion of a tree. Below, we present a definition of *space tree* that encapsulates all of the tree types mentioned in the previous chapter [28].

Definition 1. *A* space tree *on a dataset* $S \in \mathcal{R}^{N \times d}$ *is an undirected, connected, acyclic, rooted simple graph with the following properties:*

- *Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).*
- *There is one node in every space tree with no parent; this is the root node of the tree.*
- *Each point in* S *is contained in at least one node.*
- *Each node corresponds to some subset of* $\mathcal{R}^d$ *that contains each point in that node and also the subsets that correspond to each child of the node.*

There is nothing counterintuitive about this definition: a tree is a hierarchical graph structure on data, and each node corresponds to some region of the input space. As the tree is descended, the space corresponded to by each node will shrink. Each node in the tree, then, may be fully parameterized by the points it holds, the children it holds, the region of input space it corresponds to, and its parent. Despite this fairly straightforward definition, though, there are a couple important notes.

First, we are unable to use the term *space partitioning tree*, because in our definition we do not require that nodes are non-overlapping. For instance, the spill tree [56], ball trees

[48], and the cover tree [57] can each have overlapping nodes. This forces us to use the more general term *space tree*, which is also something of a nice consolation prize for the author's failed dream of becoming an astronaut.

Second, it is imperative to note that the points held by a node are *not* necessarily the same as the points held by the node's children. The *only* restriction on the sets of points held by a node's children (and by the children's children, and so forth) is that they all fall into the region of input space represented by that node. This distinction is incredibly important when talking about space trees: the term *descendant points of a node* refers to the points held by the node plus all the points held by the descendant nodes, whereas the term *points of a node* refers only to the points held in a node. This distinction will become clearer in the tree survey of Section 3.6.

Third, consider the last part of the definition: each node corresponds to some subset of $\mathcal{R}^d$. In general, trees are designed so that these subsets are geometrically easy structures to work with, such as balls, rectangles, cones, slices, and so forth. Any of these structures can usually be parameterized by only a few values; for instance, a ball only needs a center and a radius, and a rectangle only needs an origin and side lengths. This means implementation of a tree node is generally simple; it has only a list of children, a list of points, optionally a

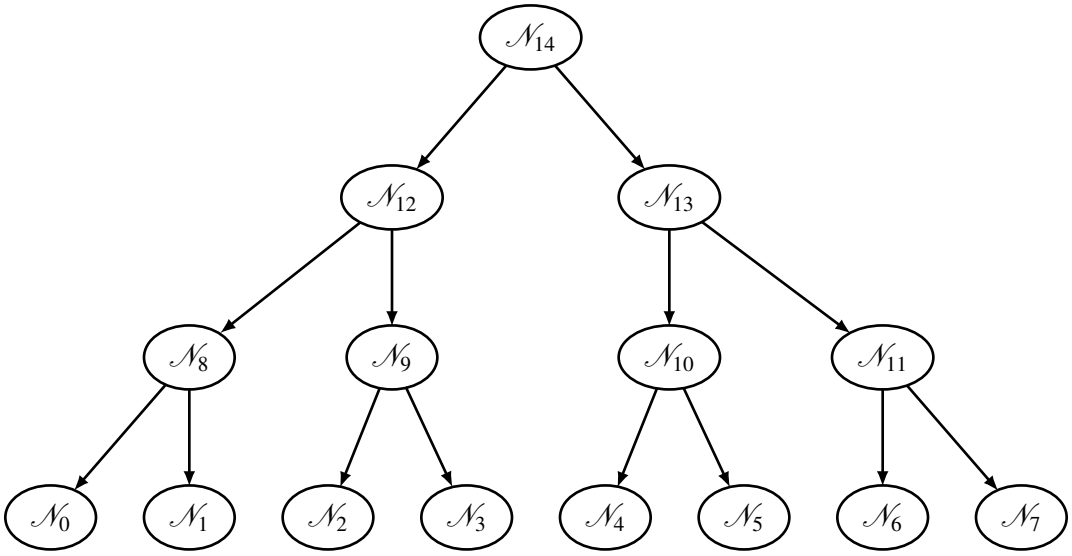
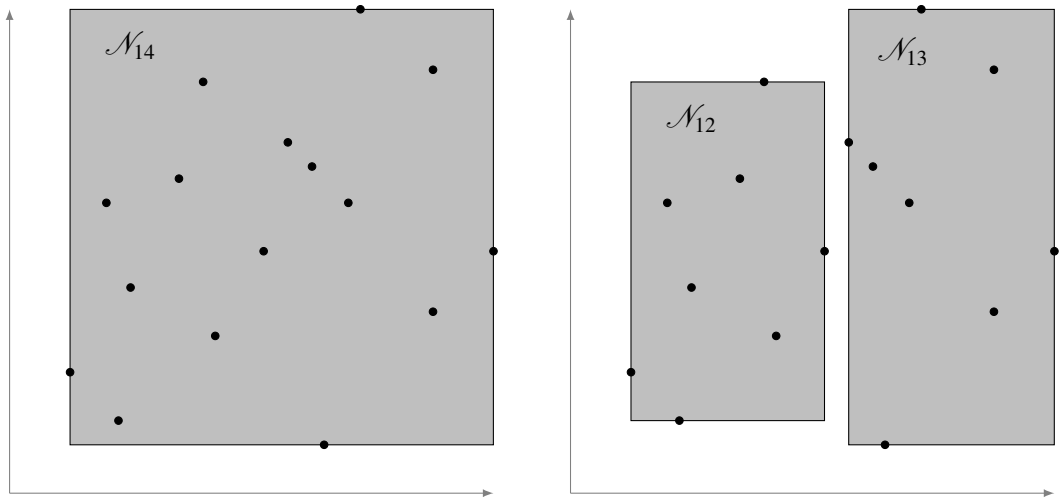
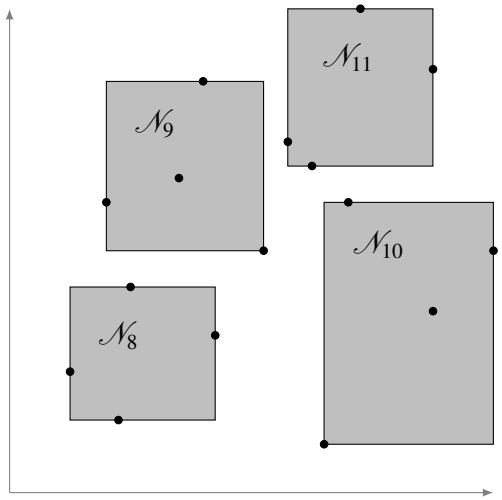


Figure 6: Abstract representation of an example *kd*-tree.

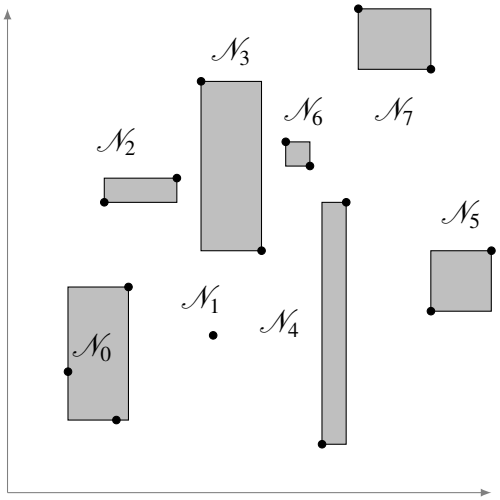


(a) Top level.

(b) Second level.



(c) Third level.



(d) Lowest level (leaves).

Figure 7: Geometric representation of the same example *kd*-tree.

parent, and whatever is necessary to parameterize its bounding shape.

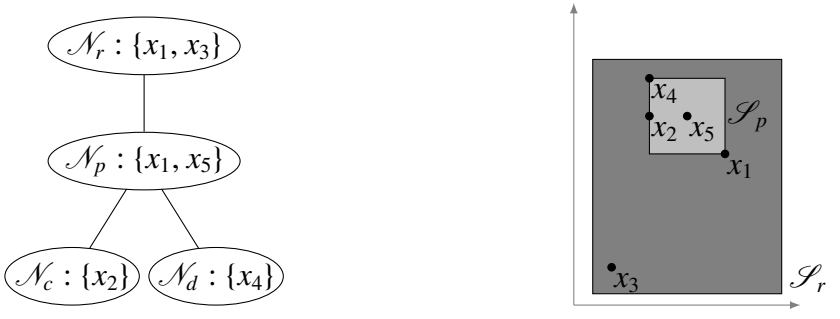
We may now revisit the *kd*-tree we presented in the previous chapter, in Figures 3, 4, and 5, and discuss this tree in the context of our definition. Figures 6 and 7 re-present the same *kd*-tree, and Figure 8 shows the parameterized representation of the children and the points held in each node. Note that *kd*-trees only hold points in the leaves, and the bounding shapes are the smallest rectangles which enclose all of the descendant points.

To further explore the possibility space for space trees, Figures 9a and 9b show another possible space tree (not a *kd*-tree). In the abstract representation of the tree given in Figure

- $\mathcal{N}_0$: children {}, points $\{p_0, p_1, p_2\}$
- $\mathcal{N}_1$: children {}, points $\{p_3\}$
- $\mathcal{N}_2$: children {}, points $\{p_4, p_5\}$
- $\mathcal{N}_3$: children {}, points $\{p_6, p_7\}$
- $\mathcal{N}_4$: children {}, points $\{p_8, p_9\}$
- $\mathcal{N}_5$: children {}, points $\{p_{10}, p_{11}\}$
- $\mathcal{N}_6$: children {}, points $\{p_{12}, p_{13}\}$
- $\mathcal{N}_7$: children {}, points $\{p_{14}, p_{15}\}$
- $\mathcal{N}_8$: children $\{\mathcal{N}_0, \mathcal{N}_1\}$, points {}
- $\mathcal{N}_9$: children $\{\mathcal{N}_2, \mathcal{N}_3\}$, points {}
- $\mathcal{N}_{10}$: children $\{\mathcal{N}_4, \mathcal{N}_5\}$, points {}
- $\mathcal{N}_{11}$: children $\{\mathcal{N}_6, \mathcal{N}_7\}$, points {}
- $\mathcal{N}_{12}$: children $\{\mathcal{N}_8, \mathcal{N}_9\}$, points {}
- $\mathcal{N}_{13}$: children $\{\mathcal{N}_{10}, \mathcal{N}_{11}\}$, points {}
- $\mathcal{N}_{14}$: children $\{\mathcal{N}_{12}, \mathcal{N}_{13}\}$, points {}

Figure 8: Parameterized representation of the same example *kd*-tree.

9a, $\mathcal{N}_r$ is the root node of the tree; it has no parent and it contains the points x_3 and x_1. The node $\mathcal{N}_p$ contains points x_1 and x_5 and has children $\mathcal{N}_c$ and $\mathcal{N}_d$ (which each have no children and contain points x_2 and x_4, respectively). Figure 9b draws the tree in the input space of the points, $\mathcal{R}^2$. The points in the tree and the subsets of input space represented by $\mathcal{N}_r$ (darker rectangle) and $\mathcal{N}_p$ (lighter rectangle) are plotted. The subsets of input space corresponding to $\mathcal{N}_c$ and $\mathcal{N}_d$ are not labeled, because those subsets are simply $\{x_2\}$ and $\{x_4\}$, respectively.



(a) Abstract representation.

(b) $\mathcal{R}^2$ representation.

Figure 9: Another example space tree.

3.4 Space tree notation

Now that we have defined a space tree and mildly explored the possibility space, we must establish standardized notation which will be used throughout the rest of the document. A quick reference table is given in Table 1, and detailed definitions are given below.

- The set of child nodes of a node $\mathcal{N}_i$ is denoted $\mathscr{C}(\mathcal{N}_i)$ or $\mathscr{C}_i$.
- The set of points held in a node $\mathcal{N}_i$ is denoted $\mathscr{P}(\mathcal{N}_i)$ or $\mathscr{P}_i$.
- The subset of input space represented by a node $\mathcal{N}_i$ is denoted $\mathscr{S}(\mathcal{N}_i)$ or $\mathscr{S}_i$.
- The set of descendant nodes of a node $\mathcal{N}_i$, denoted $\mathscr{D}^n(\mathcal{N}_i)$ or $\mathscr{D}^n_i$, is the set of nodes $\mathscr{C}(\mathcal{N}_i) \cup \mathscr{C}(\mathscr{C}(\mathcal{N}_i)) \cup \ldots$. By $\mathscr{C}(\mathscr{C}(\mathcal{N}_i))$, we mean all the children of the children of node $\mathcal{N}_i$: $\mathscr{C}(\mathscr{C}(\mathcal{N}_i)) = \{\mathscr{C}(\mathcal{N}_c) : \mathcal{N}_c \in \mathscr{C}(\mathcal{N}_i)\}$.
- The set of descendant points of a node $\mathcal{N}_i$, denoted $\mathscr{D}^p(\mathcal{N}_i)$ or $\mathscr{D}^p_i$, is the set of points $\{p : p \in \mathscr{P}(\mathscr{D}^n(\mathcal{N}_i)) \cup \mathscr{P}(\mathcal{N}_i)\}$. The meaning of $\mathscr{P}(\mathscr{D}^n(\mathcal{N}_i))$ is similar to the meaning of $\mathscr{C}(\mathscr{C}(\mathcal{N}_i))$: $\mathscr{P}(\mathscr{D}^n(\mathcal{N}_i)) = \{\mathscr{P}(\mathcal{N}_d) : \mathcal{N}_d \in \mathscr{D}^n(\mathcal{N}_i)\}$.
- The parent of a node $\mathcal{N}_i$ is denoted parent($\mathcal{N}_i$).
- The centroid of a node $\mathcal{N}_i$ is denoted centroid($\mathcal{N}_i$); this is the centroid of all descendant points of the node. Usually, this quantity is easily calculated at tree-building time and may be cached then.
- The center of a node $\mathcal{N}_i$ is denoted μ_i; this is the center of the region $\mathscr{S}_i$. This is, in general, different than the centroid; for some tree types, it is easy to calculate; for others, it is not easy.
- The furthest descendant distance for a node $\mathcal{N}_i$ and a metric $d(\cdot, \cdot)$, denoted $\lambda(\mathcal{N}_i)$ or λ_i, is defined as

Table 1: Notation for trees. See text for details.

Symbol	Description
$\mathscr{N}$	A tree node
$\mathscr{C}_i$	Set of child nodes of $\mathscr{N}_i$
$\mathscr{P}_i$	Set of points held in $\mathscr{N}_i$
$\mathscr{D}_i^n$	Set of descendant nodes of $\mathscr{N}_i$
$\mathscr{D}_i^p$	Set of points contained in $\mathscr{N}_i$ and $\mathscr{D}_i^n$
parent($\mathscr{N}_i$)	The parent of $\mathscr{N}_i$
centroid($\mathscr{N}_i$)	The centroid of all descendant points of $\mathscr{N}_i$
μ_i	Center of $\mathscr{N}_i$
λ_i	Furthest descendant distance

$$\lambda(\mathscr{N}_i) = \max_{p \in \mathscr{D}^p(\mathscr{N}_i)} d(\mu_i, p). \tag{3}$$

It is often possible to calculate $\lambda(\mathscr{N}_i)$ exactly, depending on the type of tree, or at least calculating a bound on $\lambda(\mathscr{N}_i)$ is often possible.

In general, the short notation (i.e. $\mathscr{C}_i$ instead of $\mathscr{C}(\mathscr{N}_i)$) will be used where possible, and the long notation will only be used when further clarity is required.

3.5 Bounding quantities with space trees

The real utility of a simple bounding shape comes from the ability to quickly calculate bounds on geometric quantities. In the introduction, during the discussion on single-tree nearest neighbor search, pruning was possible when the minimum distance between the node and the query point was sufficiently large. Let us now formalize this notion of minimum distance.

Definition 2. *The exact minimum distance between a node $\mathscr{N}_i$ and a point p_q is defined as*

$$d^*_{\min}(p_q, \mathscr{N}_i) := \min \left\{ d(p_q, p_j) \; \forall \; p_j \in \mathscr{D}_i^p \right\}. \tag{4}$$

In general, computing the *exact* minimum distance between a point p_q and a node $\mathscr{N}_i$ is computationally infeasible and defeats the entire purpose of trees in the first place, which

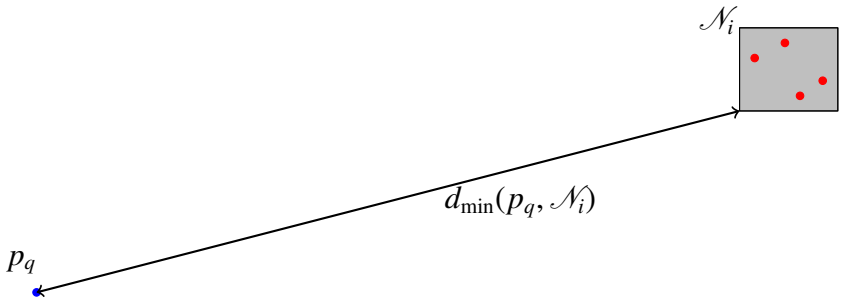


Figure 10: A bound on the minimum distance between a point p_q and a node $\mathcal{N}_i$.

is to represent the data compactly at various scales. If we have to scan every descendant point of a node to calculate the minimum distance, the entire exercise of building a tree is pointless. Fortunately, the fact that each space tree node corresponds to a subset of the input space and is often a convenient geometric shape allows us to easily place a lower bound on $d^*_{\min}(\cdot, \cdot)$; see Figure 10. Note that the distance $d_{\min}(p_q, \mathcal{N}_i)$ is a lower bound on the exhaustively calculated $d^*_{\min}(p_q, \mathcal{N}_i)$ (which is not drawn in the figure).

Given the parameters of the region represented by $\mathcal{N}_i$ (which is general we do have), calculating $d_{\min}(\cdot, \cdot)$ is a relatively trivial $O(d)$ operation. Suppose that $\mathcal{S}_i$ (unlike in the figure) is a ball of radius λ_i with center μ_i; then, we may easily calculate $d_{\min}(p_q, \mathcal{N}_i)$:

$$d_{\min}(p_q, \mathcal{N}_i) = d(p_q, \mu_i) - \lambda_i. \tag{5}$$

This calculation is easy and fast, and is often a reasonable bound for $d^*_{\min}(p_q, \mathcal{N}_i)$. For trees with different bounding shapes, the calculation can be quite different. Quadtrees, for instance, require a slightly more complex calculation, because the bounding box is a square. Similarly, *kd*-trees have hyperrectanglur bounds, which means the calculation is not as simple as taking the distance between the point p_q and the center of the node and subtracting the radius. Still, the majority of useful space trees are able to produce a bound, $d_{\min}(p_q, \mathcal{N}_i)$ in $O(d)$ time or better.

We can now generalize this point-to-node distance bound to a node-to-node distance bound.

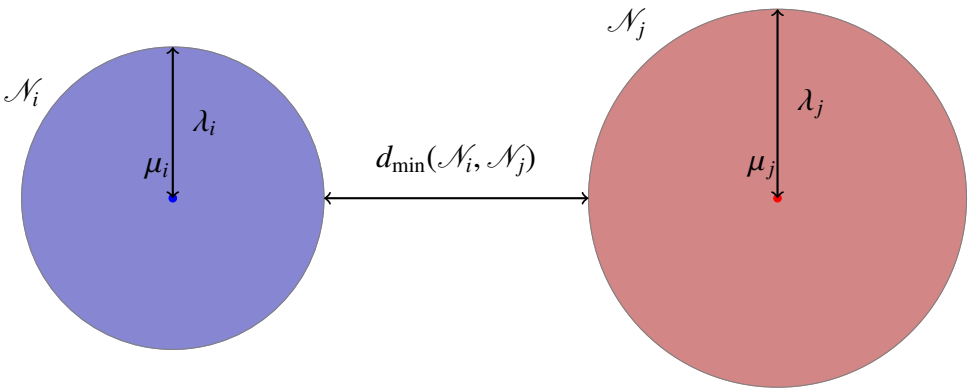


Figure 11: A bound on the minimum distance between a node $\mathscr{N}_i$ and a node $\mathscr{N}_j$.

Definition 3. *The exact minimum distance between two nodes $\mathscr{N}_i$ and $\mathscr{N}_j$ is defined as*

$$d^*_{\min}(\mathscr{N}_i, \mathscr{N}_j) := \min \left\{ d(p_i, p_j) \; \forall \; p_i \in \mathscr{D}_i^p, \; p_j \in \mathscr{D}_j^p \right\}. \tag{6}$$

Again, we may easily bound this quantity using trees. Figure 11 demonstrates this bound $d_{\min}(\mathscr{N}_i, \mathscr{N}_j)$ geometrically, in the same way as Figure 10. In the figure given, $\mathscr{S}_i$ and $\mathscr{S}_j$ are both balls with centers μ_i and μ_j and radii λ_i and λ_j, respectively. This means we can easily calculate $d_{\min}(\mathscr{N}_i, \mathscr{N}_j)$:

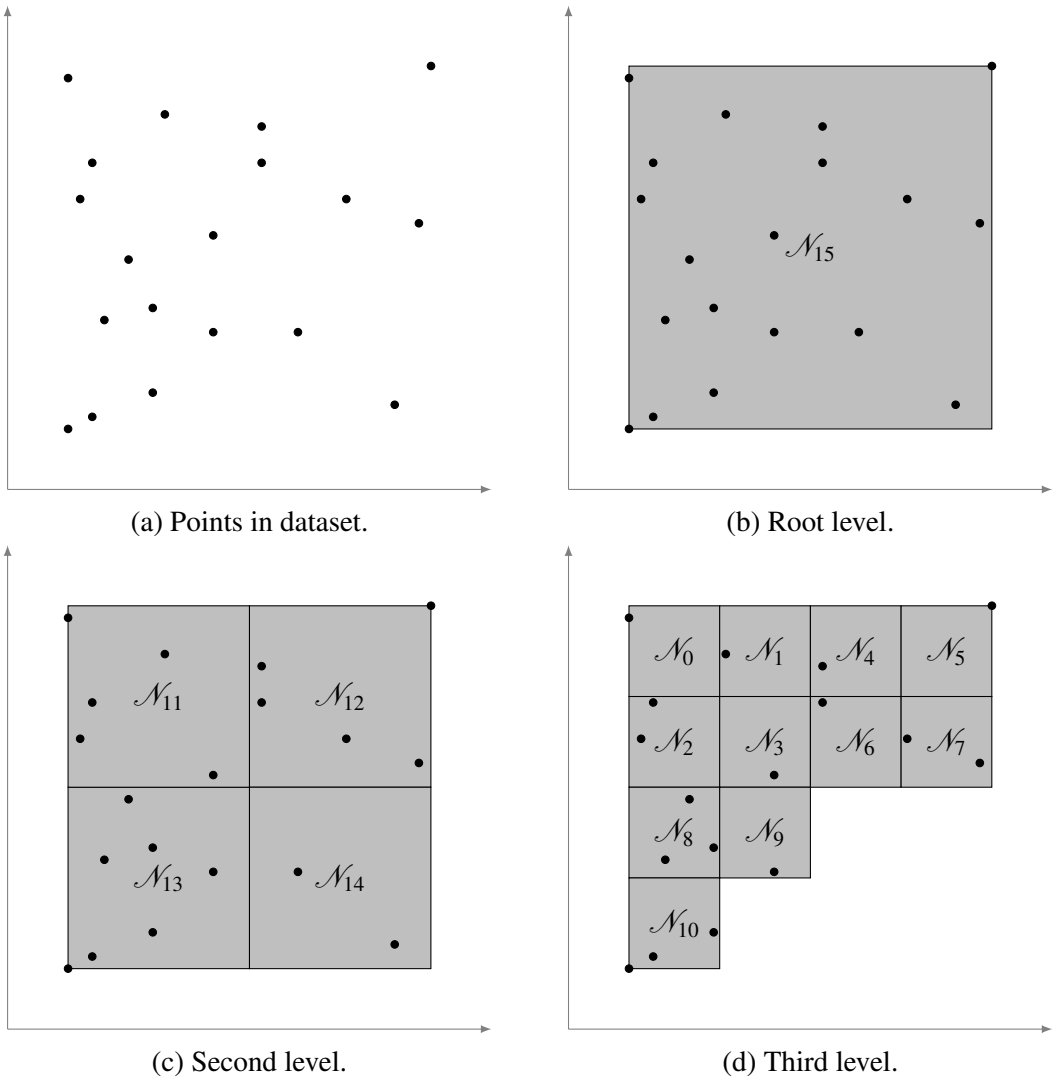
$$d_{\min}(\mathscr{N}_i, \mathscr{N}_j) = d(\mu_i, \mu_j) - \lambda_i - \lambda_j. \tag{7}$$

As with the point-to-node bound, the exact way to quickly calculate a lower bound $d_{\min}(\cdot, \cdot)$ varies across tree types.

It is easy to extend the intuition used to define $d_{\min}(\cdot, \cdot)$ to other quantities, like $d_{\max}(\cdot, \cdot)$. These bounds provide useful summarization of the data points contained in a node, and fast evaluation of these bounds is paramount to any of the tree-based strategies discussed here or in related works.

3.6 A quick survey of some space trees

In order to demonstrate the utility of the space tree abstraction, let us now consider popular types of trees and show how they are described generally as space trees. The descriptions



(a) Points in dataset.

(b) Root level.

(c) Second level.

(d) Third level.

Figure 12: Three levels of an example quadtree with a leaf size of 3.

here do not consider how the tree is built in-depth, since the tree-building procedure is somewhat unimportant with respect to how the tree fits into the space tree abstraction. There are many more types of trees, as I alluded to in the introduction; understanding these as space trees is often straightforward and can be done using the same type of reasoning as in the following subsections.

3.6.1 The quad-tree, octree, and hyperoctree

The quad-tree [29], octree [30], and hyperoctree [86] are all the same tree, just specialized for different dimensionalities. A quad-tree lives in two dimensions and each node has up to

Table 2: Properties of hyperoctrees.

Quantity	Description	Value for hyperoctrees
$\mathscr{C}_i$	children of $\mathscr{N}_i$	$\emptyset$ for leaves, $\lvert\mathscr{C}_i\rvert \leq 2^d$ otherwise
$\mathscr{P}_i$	points of $\mathscr{N}_i$	$\emptyset$ for non-leaves, all points in $\mathscr{S}_i$ otherwise
$\mathscr{S}_i$	region of $\mathscr{N}_i$	hypercube with side-length l
μ_i	center of $\mathscr{N}_i$	center of hypercube $\mathscr{S}_i$
λ_i	furthest desc. distance	$\sqrt{d(l/2)^2}$

Property	Value for hyperoctrees
Can nodes overlap?	No.
Points in multiple nodes?	No.
Points only in leaves?	Yes.
Leaves hold all points?	Yes.

four children; an octree lives in three dimensions and each node has up to eight children; a hyperoctree living in d dimensions has up to 2^d children.

Building a hyperoctree (which is the term I will use from here) in dimension d involves finding a hypercube that contains all the data. This is the root of the tree. This hypercube is then split into 2^d hypercubes with side lengths equal to half of the root. Any hypercubes that contain no points in the dataset are not created. Thus, the root may contain up to 2^d children. This splitting procedure continues recursively until a node contains some specified number of points (the *leaf size*). Figure 12 shows a quadtree (that is, a hyperoctree with dimension 2) with a leaf size of 3; this is the same quadtree shown in the previous chapter.

In general, most implementations of hyperoctrees store points only in the leaves; that is, for some node $\mathscr{N}_i$, $\mathscr{P}_i = \{\}$ unless $\mathscr{N}_i$ is a leaf. With our description complete, we may now summarize characteristics of the hyperoctree in Table 2.

Because of the huge number of children a hyperoctree has at high dimensions, hyperoctrees are often a bad choice past $d > 3$ or so. Often, normalizing a dataset so each dimension has unit variance is a good choice, because of the restriction that hyperoctrees correspond to hypercubes.

Table 3: Properties of *kd*-trees.

Quantity	Description	Value for *kd*-trees
$\mathscr{C}_i$	children of $\mathscr{N}_i$	$\emptyset$ for leaves, $\lvert\mathscr{C}_i\rvert = 2$ (left and right) otherwise
$\mathscr{P}_i$	points of $\mathscr{N}_i$	$\emptyset$ for non-leaves, all points in $\mathscr{S}_i$ otherwise
$\mathscr{S}_i$	region of $\mathscr{N}_i$	hyperrectangle enclosing all descendant points
μ_i	center of $\mathscr{N}_i$	center of hyperrectangle $\mathscr{S}_i$
λ_i	furthest desc. distance	distance between μ_i and any corner of $\mathscr{S}_i$

Property	Value for *kd*-trees
Can nodes overlap?	No.
Points in multiple nodes?	No.
Points only in leaves?	Yes.
Leaves hold all points?	Yes.

3.6.2 The *kd*-tree

The *kd*-tree [31] presents a better solution for higher-dimensional data by only allowing two children per node and allowing hyperrectangle bounds, instead of hypercube bounds.

A *kd*-tree is built by first finding an enclosing hyperrectangle for all of the data points; this is the root. Then, a dimension is chosen to split on (often, this is the dimension with maximum variance). A split value is then chosen; this may be the median, mean, or midpoint of the data in the chosen dimension. Points with value in the chosen dimension less than or equal to the split value will be descendants of the *left node*; points with value greater than the split value will be descendants of the *right node*. This procedure is continued recursively, until a node contains some specified number of points (again, the *leaf size*). Figure 7, in Section 3.3, shows an example *kd*-tree.

As with hyperoctrees, most implementations of *kd*-trees store points only in the leaves. A summary of the characteristics of *kd*-trees is given in Table 3.

3.6.3 The ball tree

The ball tree is not a specific type of tree, but encompasses many different types of trees with similar characteristics[1]; for example, Omohundro describes five different construction

[1]Our discussion here assumes ball trees to be quite similar to *kd*-trees. Other authors may take 'ball tree' to mean something else entirely, but we stick to the general definition settled on by Gray and coauthors [61] and implemented in the **mlpack** machine learning library [87].

Table 4: Properties of ball trees.

Quantity	Description	Value for ball trees
$\mathscr{C}_i$	children of $\mathscr{N}_i$	$\emptyset$ for leaves, $\lvert\mathscr{C}_i\rvert = 2$ (left and right) otherwise
$\mathscr{P}_i$	points of $\mathscr{N}_i$	$\emptyset$ for non-leaves, all points in $\mathscr{S}_i$ otherwise
$\mathscr{S}_i$	region of $\mathscr{N}_i$	a ball enclosing all descendant points
μ_i	center of $\mathscr{N}_i$	center of ball $\mathscr{S}_i$
λ_i	furthest desc. distance	radius of ball $\mathscr{S}_i$

Property	Value for ball trees
Can nodes overlap?	Yes.
Points in multiple nodes?	No.
Points only in leaves?	Yes.
Leaves hold all points?	Yes.

algorithms for balltrees [48]. Roughly, the ball tree may be understood as an analog of the *kd*-tree, where each node has a *left* and *right* child. However, instead of each node being represented by a hyperrectangle, it is instead represented by a ball.

Each ball corresponding to a node should ideally be the smallest ball that encloses all of the node's descendant points, but the minimum enclosing ball problem is known to be difficult, with a simple implementation finding the minimum enclosing ball over n points taking $O(n^4)$ time. Fortunately, faster algorithms do exist, such as the $O(n \log n)$ algorithm of Shamos and Hoey [88] and the later $O(n)$ linear programming algorithm of Megiddo, Zemel, and Hakimi [89]. Unfortunately, Megiddo's algorithm is impractical for large d, with a running time of $O((d+1)(d+1)!n)$; thus, there exist numerous alternate strategies to provide approximate bounding spheres [90, 91, 92, 93]. Nonetheless, even with these accelerated algorithms, quickly computing a 'good' minimum enclosing ball approximation is a difficult challenge for ball trees.

With a good minimum enclosing ball, though, the computation of minimum distances between nodes is simple, as in Equation 7 from the last section. The general structure of ball trees is the same as *kd*-trees, with two children per non-leaf node, and points only held in the leaves. Properties of ball trees are given in Table 4. Note that although the two children of a ball tree node may overlap, no points are held in *both* the left and right child.

3.6.4 The metric tree / vantage-point tree

The metric tree, developed by Uhlmann in 1991 [51], or the vantage point tree, developed by Yianilos in 1993 [50], turn out to be the same tree structure. I will use the "vantage-point tree" name here because I find it to be more descriptive of the tree type.

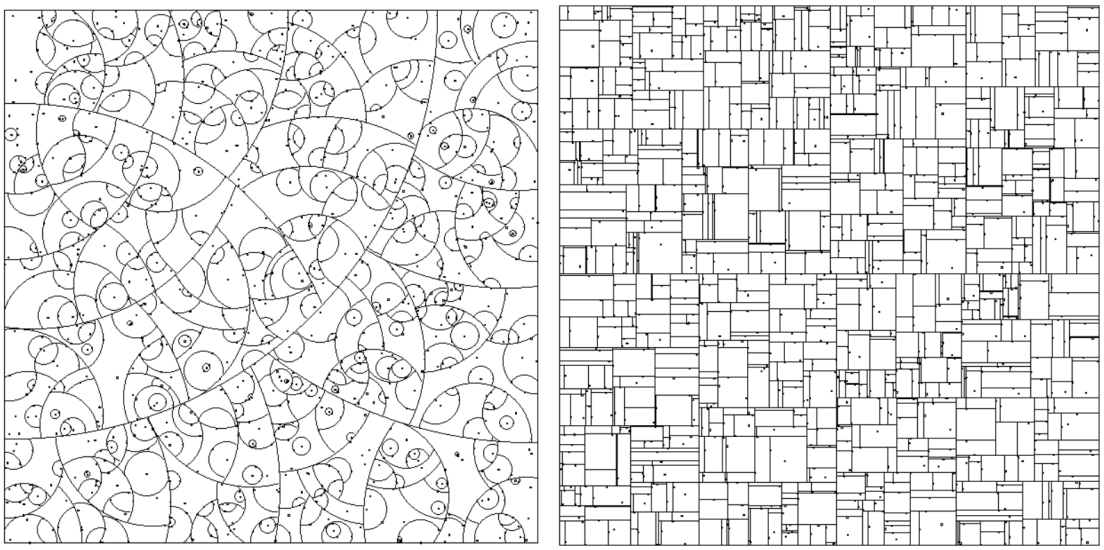
Instead of each node representing a different configuration of the same general shape, each vantage point tree node $\mathcal{N}_i$ has a *near child* (also called left or inside child) and a *far child*[2] (also called right or outside child), and is centered at a point p_i. The near child corresponds to the ball centered at p_i with some radius r, and the far child corresponds to the ball slice centered at p_i with minimum radius r and maximum radius λ_i.

One way of constructing a vantage point tree (the "simplest" vp-tree, according to Yianilos [50]) is as follows. A ball that encloses the entire dataset and is centered at some point p_i is chosen, usually by some type of random sampling procedure; this corresponds to the root node. Then, the median distance of other points from p_i is chosen as the radius r; points with distance less than r from p_i go into the close child, and points with distance greater than or equal to r from p_i go into the far child. This process is repeated recursively, but the "vantage point" p_i is not passed into its children. Thus, this tree is different in that not all points are held at the leaves, and the leaves do not hold every point in the dataset.

Construction of the tree may continue recursively until a node contains only one point. It is possible to modify the construction algorithm to stop splitting when a node contains some specified number of points (as with earlier trees, the leaf size).

The unusual shape of the near and far child means that the space decomposition of a vantage-point tree looks significantly different than the axis-aligned hyperrectangles of the *kd*-tree. Figure 13a and Figure 13b illustrate this difference; they are reprinted from Yianolis' original work [50]. Table 5 lists characteristics of the vantage-point tree, in the same format as previous tables.

[2]This is not related to Starchild, the divine being who comes to earth to bring Funk to humanity, according to George Clinton and his associates in the bands Parliament and Funkadelic. Despite this, Starchild continues to be a strong influence on the work of the author.



(a) vp-tree decomposition of a dataset (from [50]).

(b) *kd*-tree decomposition of the same dataset (from [50]).

Figure 13: *kd*-tree and vp-tree space decompositions.

Table 5: Properties of vp-trees.

Quantity	**Description**	**Value for vp-trees**
$\mathscr{C}_i$	children of $\mathscr{N}_i$	$\emptyset$ for leaves, $\lvert\mathscr{C}_i\rvert = 2$ (near and far) otherwise
$\mathscr{P}_i$	points of $\mathscr{N}_i$	vantage point p_i
$\mathscr{S}_i$	region of $\mathscr{N}_i$	a ball centered at p_i
μ_i	center of $\mathscr{N}_i$	center of ball $\mathscr{S}_i$, which is p_i
λ_i	furthest desc. distance	radius of ball $\mathscr{S}_i$

Property	**Value for ball trees**
Can nodes overlap?	Yes.
Points in multiple nodes?	No.
Points only in leaves?	No.
Leaves hold all points?	No.

3.6.5 The cover tree

The cover tree, more recently proposed in 2006 [57], is the most complex tree type we will consider here. Its complexity stems from its theoretical utility, which we will discuss much further in detail later in the paper. This subsection only serves as a basic introduction to the basic properties of the tree and how it fits into the space tree abstraction.

When building a cover tree, we assume only that we have some dataset S and some

metric $d(\cdot, \cdot)$; so, like the vantage point tree, the space need not be Euclidean or even representable. Like the vantage point tree, each node in a cover tree is parameterized by a center point p_i and a radius $\lambda_i \leq 2^{s_i}$, where s_i is the integer *scale* of the node. The root of the tree has the largest scale, and the leaves (with scale $-\infty$) have the smallest scale. However, the cover tree also satisfies some other invariants. I have yet to find a better summary than the original provided by the authors [57], so I will simply quote their words:

> A *cover tree* $\mathscr{T}$ on a dataset S is a leveled tree where each level is a "cover" for the level beneath it. Each level is indexed by an integer scale s_i which decreases as the tree is descended. Every *node* in the tree is associated with a point in S. Each *point* in S may be associated with multiple nodes in the tree; however, we require that any point appears at most once in every level. Let C_{s_i} denote the set of points in S associated with the nodes at level s_i. The cover tree obeys the following invariants for all s_i:
>
> - *(Nesting).* $C_{s_i} \subset C_{s_i - 1}$. This implies that once a point $p \in S$ appears in C_{s_i} then *every* lower level in the tree has a node associated with p.
> - *(Covering tree).* For every $p_i \in C_{s_i - 1}$, there exists a $p_j \in C_{s_i}$ such that $d(p_i, p_j) < 2^{s_i}$ and the node in level s_i associated with p_j is a parent of the node in level $s_i - 1$ associated with p_i.
> - *(Separation).* For all distinct $p_i, p_j \in C_{s_i}$, $d(p_i, p_j) > 2^{s_i}$.

As a consequence of this definition, if there exists a node $\mathscr{N}_i$, containing the point p_i at some scale s_i, then there will also exist a self-child node $\mathscr{N}_{ic}$ containing the point p_i at scale $s_i - 1$ which is a child of $\mathscr{N}_i$. In addition, every descendant point of the node $\mathscr{N}_i$ is contained within a ball of radius 2^{s_i+1} centered at the point p_i; therefore, the furthest descendant distance λ_i is bounded by 2^{s_i+1} and the center of the node μ_i is the point p_i.

Note that the cover tree may be interpreted as an infinite-leveled tree, with C_∞ containing only the root point, $C_{-\infty} = S$, and all levels between defined as above. Beygelzimer

Table 6: Properties of cover trees.

Quantity	Description	Value for cover trees
$\mathscr{C}_i$	children of $\mathscr{N}_i$	$\emptyset$ for leaves, $\lvert\mathscr{C}_i\rvert$ potentially large otherwise
$\mathscr{P}_i$	points of $\mathscr{N}_i$	one point, p_i
$\mathscr{S}_i$	region of $\mathscr{N}_i$	a ball centered at p_i with radius 2^{s_i+1}
μ_i	center of $\mathscr{N}_i$	center of ball $\mathscr{S}_i$, which is p_i
λ_i	furthest desc. distance	2^{s_i+1}

Property	Value for ball trees
Can nodes overlap?	Yes.
Points in multiple nodes?	Yes.
Points only in leaves?	No.
Leaves hold all points?	Yes.

et al. [57] find this representation (which they call the *implicit* representation) easier for description of their algorithms and some of their proofs. But clearly, this is not suitable for implementation; hence, there is an *explicit* representation in which all nodes that have only a self-child are coalesced upwards (that is, the node's self-child is removed, and the children of that self-child are taken to be the children of the node). In this work, we consider *only* the explicit representation of a cover tree.

Thus, the major structural differences from any tree we have considered in-depth so far is that points may exist at multiple levels of the tree. We can encapsulate the primary properties of the cover tree in Table 6.

3.7 Traversals and problem-specific rules

Now that we have defined a tree in an abstract sense, we no longer need to think about the individual properties of trees and can consider them by using our abstract *space tree* definition. The definitions we present here formalize and abstract the traversal strategies used by the numerous single-tree and dual-tree algorithms.

It is easier to start with the single-tree traversal definitions.

Definition 4. *A **single-tree traversal** is a process that, given a space tree, will visit each node in that tree once and perform a computation on any points contained within the node*

that is being visited; call this computation `BaseCase()`.

As an example, the standard depth-first traversal or breadth-first traversal are single-tree traversals. From a programming perspective, the computation in the single-tree traversal can be implemented with a simple callback `BaseCase(point)` function. This allows the computation to be entirely independent of the single-tree traversal itself. As an example, a simple single-tree algorithm to count the number of points in a given tree would increment a counter variable each time `BaseCase(`p_i`)` was called, where p_i is some point in the node currently being visited. Note that at each node, `BaseCase()` is only called on those points in $\mathscr{P}_i$, not all descendant points of the node. So, as long as the tree type being used satisfied the condition that each point is contained in only one node, this example algorithm to count the number of points would return the correct result.

However, this concept of a single-tree traversal by itself is not very useful; without pruning branches, no computations can be avoided. Thus, we must now introduce a mechanism for pruning.

Definition 5. *A* ***pruning single-tree traversal*** *is a process that, given a space tree, will visit nodes in the tree and perform a computation to assign a score to that node; call this computation* `Score()`. *If the score is* ∞*, the node is "pruned" and none of its descendants will be visited; otherwise, a computation is performed on any points contained within that node; call that computation* `BaseCase()`. *If no nodes are pruned, then the traversal will visit each node in the tree once.*

Clearly, a pruning single-tree traversal that does not prune any nodes is just a single-tree traversal. A pruning single-tree traversal can be implemented with two callbacks: `BaseCase()` and `Score()`. This allows both the point-to-point computation and the scoring to be entirely independent of the traversal. Thus, single-tree branch-and-bound algorithms can be expressed in a tree-independent manner.

Below is a simple `BaseCase()` function (Algorithm 4) and `Score()` function (Algorithm 5) that will print "Hello!" once for each point in a reference tree $\mathscr{T}_r$ that has distance

less than or equal to 1 from a given query point p_q.

Algorithm 4 `BaseCase()` for hello-printing nonsense example algorithm.

```
1: Input: query point p_q, reference point p_r
2: if p_r not already visited with query point p_q then
3:     if d(p_q, p_r) ≤ 1 then
4:         print "Hello!"
```

Algorithm 5 Single-tree `Score()` for hello-printing nonsense example algorithm.

```
1: Input: query point p_q, node N_i from tree T
2: if d_min(p_q, N_i) > 1 then
3:     {Prune the node; it is too far away.}
4:     return ∞
5: else
6:     {Recursion order does not matter here; we just return an arbitrary finite value.}
7:     return 867.5309
```

The `Score()` function prunes away a branch of the tree if the given node is sufficiently far away from the point p_q (that is, if the minimum distance between p_q and any descendant point of $\mathcal{N}_i$ is greater than 1). There are two details in the algorithm that deserve further discussion.

The first is the conditional "**if** p_i not already visited with query point p_q **then**". For some trees, we know that points cannot be duplicated across nodes (the cover tree is the only exception we have considered here in any detail, but the spill tree also can duplicate points). So in those cases where we know that points cannot be duplicated, the check is simply unnecessary. In the case of the more complex cover tree, the check can be restated in an easy-to-compute manner: "if the parent of $\mathcal{N}_i$ does not hold p_i".

The second detail is the value that `Score()` returns. The definition of a pruning single-tree traversal only requires that ∞ signifies that the node should be pruned, but when the node is not pruned, there is no restriction on what `Score()` should return. In practice, though, many single-tree algorithms (and dual-tree algorithms—more on this shortly) benefit from a *prioritized* recursion. One example is nearest neighbor search: searches are faster when you recurse first into nodes $\mathcal{N}_i$ with smaller $d_{\min}(p_q, \mathcal{N}_i)$, because this is more

likely to decrease the distance between p_q and its candidate nearest neighbor, which will in turn allow more pruning during the search [66].

In the case of our simple algorithm above, the recursion order makes no difference as far as pruning is concerned, and we are unconcerned with the order in which we receive our (potentially many) greetings; thus, I have chosen an arbitrary value to return in accordance with the ideas of Tommy Tutone. Any other finite value could work just fine too.

Now, let us extend our general definition to the dual-tree case.

Definition 6. *A **dual-tree traversal** is a process that, given two space trees $\mathscr{T}_q$ (query tree) and $\mathscr{T}_r$ (reference tree), will visit every combination of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ once, where $\mathscr{N}_q \in \mathscr{T}_q$ and $\mathscr{N}_r \in \mathscr{T}_r$. At each visit $(\mathscr{N}_q, \mathscr{N}_r)$, a computation is performed between each point in $\mathscr{N}_q$ and each point in $\mathscr{N}_r$; call this computation* `BaseCase()`.

The primary difference between the single-tree definition and the dual-tree definition is that instead of visiting a single node $\mathscr{N}_i$ at a time, we are visiting a *combination* of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ where $\mathscr{N}_q$ is the query node and $\mathscr{N}_r$ is the reference node. As with the single-tree traversal, if the tree type is such that each point is held in only one node, then any dual-tree traversal will call `BaseCase()` once on each combination of query point and reference point.

Again, we can introduce the notion of a `Score()` function for pruning.

Definition 7. *A pruning dual-tree traversal is a process that, given two space trees $\mathscr{T}_q$ (the query tree, built on the query set S_q) and $\mathscr{T}_r$ (the reference tree, built on the reference set S_r), will visit combinations of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ such that $\mathscr{N}_q \in \mathscr{T}_q$ and $\mathscr{N}_r \in \mathscr{T}_r$ no more than once, and call a function* `Score(`$\mathscr{N}_q$`, `$\mathscr{N}_r$`)` *to assign a score to that node. If the score is ∞, the combination is pruned and no combinations $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ such that $\mathscr{N}_{qc} \in \mathscr{D}_q^n$ and $\mathscr{N}_{rc} \in \mathscr{D}_r^n$ are visited. Otherwise, for every combination of points (p_q, p_r) such that $p_q \in \mathscr{P}_q$ and $p_r \in \mathscr{P}_r$, a function* `BaseCase(`p_q`, `p_r`)` *is called. If no node combinations are pruned during the traversal,* `BaseCase(`p_q`, `p_r`)` *is called at least once on each combination of $p_q \in S_q$ and $p_r \in S_r$.*

Algorithm 6 Dual-tree `Score()` for hello-printing nonsense example algorithm.

1: **Input:** query node $\mathscr{N}_q$ from tree $\mathscr{T}_q$, node $\mathscr{N}_r$ from tree $\mathscr{T}_r$
2: **if** $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > 1$ **then**
3: {Prune the node combination; they are sufficiently far apart.}
4: **return** ∞
5: **else**
6: {Recursion order does not matter here; we just return an arbitrary finite value.}
7: **return** 1968

Though this definition is quite complex, it is a generalization of the single-tree traversal to the dual-tree situation. The `BaseCase()` function is identical; it compares two points. But the `Score()` function is different: in the dual-tree setting, it compares a query node and a reference node, whereas in the single-tree setting, it compares a query point and a reference node.

We may revisit the `Score()` function for our greeting algorithm from earlier, and extend it to the dual-tree scenario. Now we assume that (for some unclear reason) we have an entire query set S_q and a reference set S_r—as opposed to a single query point p_q and reference set S_r—and we wish to print "Hello!" once for each pair (p_{qi}, p_{ri}) where $p_{qi} \in S_q$ and $p_{ri} \in S_r$ such that $d(p_{qi}, p_{ri}) \leq 1$. We may use a `BaseCase()` and `Score()` function to describe a dual-tree algorithm to perform this task. The `BaseCase()` has already been given in Algorithm 4—it generalizes from the single-tree case without modification. `Score()` is given in Algorithm 6.

In the dual-tree `Score()` function, we are able to prune for many query points at once: if $\mathscr{N}_q$ and $\mathscr{N}_r$ are sufficiently far apart (specifically, if the minimum distance between the two nodes is greater than 1), then no combination of descendant points between $\mathscr{D}_q^p$ and $\mathscr{D}_r^p$ can have distance less than 1, and thus they do not need to be visited. Again, in this problem, recursion order does not matter, so I have selected another arbitrary value, this time influenced by the year Miles Davis first released two landmark albums incorporating electric instruments, thus (in part) paving the direction towards jazz-rock fusion and later exciting experimentation.

Algorithm 7 `DepthFirstTraversal(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)`.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: none

{Check to see if combination can be pruned.}
if `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` $= \infty$ **then**
  return

{Perform base cases for combinations of points held in the nodes.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    `BaseCase(`p_q`,` p_r`)`

{Recurse into combinations of children.}
for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
  for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
    `DepthFirstTraversal(`$\mathscr{N}_{qc}$`,` $\mathscr{N}_{rc}$`)`

An example pruning dual-tree traversal is given in Algorithm 7. This traversal is a dual depth-first traversal, and is generalized directly to the arbitrary space tree case from Gray's earlier work [61, 64].

The traversal is straightforward: when visiting a node combination $(\mathscr{N}_q, \mathscr{N}_r)$, we first check to see if it can be pruned, using the `Score()` function. Then, we perform base cases between the points held in $\mathscr{N}_q$ and $\mathscr{N}_r$ using the `BaseCase()` function. Lastly, we recurse into all child combinations of $\mathscr{N}_q$ and $\mathscr{N}_r$. It is not difficult to refactor this traversal into a dual breadth-first traversal or a combined traversal which is depth-first in the query tree and breadth-first in the reference tree (or vice versa).

Importantly, note that our definition of traversals here are *problem-independent*: although certain traversals will be better choices for certain problems, our definition is sufficiently general that no piece of the problem is wrapped into the traversal itself.

3.8 A meta-algorithm to produce a dual-tree algorithm

With each piece of dual-tree algorithms defined, we may propose a meta-algorithm to create a dual-tree algorithm from the pieces:

Given a type of space tree, a pruning dual-tree traversal, a `BaseCase()` *function, and a* `Score()` *function, use the pruning dual-tree traversal with the given* `BaseCase()` *and* `Score()` *functions on two space trees $\mathscr{T}_q$ (built on S_q) and $\mathscr{T}_r$ (built on S_r).*

This modular way of viewing tree-based algorithms has several useful immediate applications. The first is implementation. Given a tree implementation and a dual-tree traversal implementation, all that is required is `BaseCase()` and `Score()` functions. Thus, code reuse can be maximized, and new algorithms can be implemented simply by writing two new functions. More importantly, the code is now modular. **mlpack** [87], the subject of the next chapter, which is written in C++, uses templates to accomplish this.

The next advantage of the abstraction is clear when we consider all of the papers which do not use this abstraction. One example, March's dual-tree Borůvka algorithm for calculating minimum spanning trees [68], provides a great example. The paper contains two dual-tree algorithms: one for *kd*-trees and one for cover trees. Each algorithm given is quite different and it is not easy to see their similarities. Using our meta-algorithm, any of these tree-based algorithms can be expressed with less effort—especially for more complex trees like the cover tree—and in a more general sense.

In addition, correctness proofs for our algorithms tend to be quite simple. These proofs for each algorithm here can be given in two simple sub-proofs: *(1)* prove the correctness of `BaseCase()` when no prunes are made, and *(2)* prove that `Score()` does not prune any subtrees on which the algorithm's correctness depends.

The logical split of base case, pruning rule, tree type, and traversal can also be advantageous. As one example, the split distills the pruning rule in `Score()` to its essence, easing the cognitive load of understanding the algorithm and allowing more general pruning observations to be made. In the case of nearest neighbor search, this has led to a new tighter pruning bound, which is discussed far later in the document, in Section 7.1.

More importantly, this logical split allows us to consider each part of dual-tree algorithms individually, and make improvements to each component individually. These improvements are then general enough to apply to many dual-tree algorithms, not just one: if we develop a new type of tree, we do not need to re-tune every existing dual-tree algorithm to work with the new type of tree. Instead, we can simply plug in existing `BaseCase()` and `Score()` functions for a given problem into some type of pruning dual-tree traversal, and use our new type of tree, and the algorithm will work as intended. This point in particular is the entire crux of this thesis: we will show the utility of this logical split and see how the entire class of dual-tree algorithms can be improved by improving each of the modular pieces.

CHAPTER 4

MLPACK: A FLEXIBLE C++ FRAMEWORK

Over the past six years, I have been fortunate to lead the **mlpack** machine learning library development and maintenance effort. The project has changed much since its original inception in 2007 (as FASTLIB/MLPACK) when it was a storehouse for implementations of new algorithms, and continues to change as the project gains momentum. At the time of this writing, in 2015, **mlpack** nears 30k downloads, has almost 40 contributors, and contains nearly 100k lines of code.

This chapter provides an introduction to the reasons motivating **mlpack**, the Armadillo linear algebra library which makes up the core of **mlpack**, the template metaprogramming techniques used to obtain fast, generic machine learning implementations, and how tree-independent dual-tree algorithms are currently implemented in **mlpack**[1].

4.1 A survey of the landscape of machine learning libraries

As with virtually any niche, the landscape of machine learning libraries (both before and after **mlpack**'s introduction) is scattered, smothered, and diced: there are numerous libraries but few general-purpose libraries. Libraries generally are specific to one language or environment (i.e. Java, R, MATLAB); in addition, there are very many one-off tools for single purposes—for example, `libsvm` is specifically for SVMs [94]. There are very many dead libraries which are no longer maintained, and there are a handful of libraries that try to unify the scattered landscape by using many single-purpose libraries to build a general-purpose library. There are also numerous attempts at large-scale distributed machine learning libraries such as those built on Spark [95] and Apache Mahout [96][2].

[1] Depending on how long it has been since the publication of this document, the information here may be quite out of date; for exact and up-to-date API details, you should refer to the **mlpack** website at `http://www.mlpack.org`.

[2] The concern here is not distributed systems, so we will not focus on distributed machine learning libraries. It is an interesting direction for future **mlpack** development, though.

It is becoming increasingly difficult, though, to see things this way because the machine learning and data science communities seem to have settled on `scikit-learn`, a Python toolkit that implements a vast variety of standard machine learning techniques [97]. `scikit-learn` is not the only Python toolkit for machine learning; there is also MLPy (now dead) [98], the similarly named PyML [99], Elefant (also dead) [100], PyBrain [101], Theano [102], and numerous others. Nonetheless, `scikit-learn` is by far the most widely used of these libraries, and is paralleled in popularity perhaps only by R.

But one large problem with the choice of Python or R as a language (or Java, like Weka [103]) is that it is often more difficult to achieve runtime efficiency. In Python, one must often write Cython, a language extension of Python that compiles directly to C [104]. In R, the `Rcpp` package is often used to call out to fast C++ implementations [105]. MATLAB provides the `mex` compiler so that fast C++ code can be called from inside MATLAB code.

On the other hand, it is colloquially believed that the fastest code tends to be lower-level code—FORTRAN, C, or C++. This is because the low-level thinking necessary to write effective code in these languages often forces consideration of processor architecture, cache effects, linear memory accesses, and so forth; in high-level languages like Python or MATLAB or R, these details are all hidden from the user and thus it is more difficult to write fast code. Based on this reasoning, if we are aiming to produce fast code, eschewing R and Python in favor of a lower-level language is a reasonable choice. Indeed, this choice was made by the Vowpal Wabbit online learning library [106], the SHOGUN Machine Learning Toolbox [107], and the Shark machine learning library [108]; all three of these libraries are written in C++. Vowpal Wabbit in particular is known for its speed, and this stems in part from the choice of C++ as language. Another particular advantage of C++ is the availability of *template metaprogramming*, a technique that can allow us to write generic, reusable code, often without runtime performance penalties for that genericity [109].

Unfortunately, at the time of the inception of **mlpack**, there was no general-purpose

library for tree-based algorithms, and there were no open-source implementations of dual-tree algorithms at all. Hence, **mlpack** development commenced, and now centers around the following simple list of goals:

- to implement **scalable, fast** machine learning algorithms,
- to design an **intuitive, consistent, and simple** API for non-expert users,
- to implement a **variety** of machine learning methods, and
- to provide **cutting-edge** machine learning algorithms unavailable elsewhere.

Next comes a discussion of how we have achieved each of these goals during development.

4.2 The Armadillo linear algebra library

For linear algebra, **mlpack** uses the Armadillo linear algebra library [110], which depends on heavy use of template metaprogramming techniques to achieve speed and ease of use. Armadillo expressions are painless and easy-to-understand; consider the example program below that, after generating some randomly distributed matrix X, computes the inverse of $X^T X + 3I$:

```
#include <armadillo>

using namespace arma;

int main()
{
  mat X;
  X.randu(50, 50); // Size will be 50x50.

  mat Y = inv(X.t() * X + 3 * eye<mat>(50, 50));
}
```

This syntax is based on the syntax of MATLAB or Octave, but in general, linear algebra expressions in Armadillo execute far faster than their MATLAB or Octave counterparts.

The speed of Armadillo comes from two places: its dependence on LAPACK and BLAS (or suitable replacements such as MKL, ACML, or OpenBLAS), and its ability to remove the need for temporary objects via template metaprogramming. Let us discuss this second point in greater detail. Consider the following Armadillo expression:

```
mat Y = (A + B + C) * D;
```

where `A`, `B`, `C`, and `D` are each matrices. If one were to write this expression in MATLAB or Octave, the program would first add `A` and `B`, storing the result in a temporary matrix. Then, `C` would be added to this temporary matrix, resulting in another temporary. Finally, this second temporary matrix would be multiplied with `D` and stored in the output matrix `Y`[3].

All of these temporaries are avoidable with C++ thanks to operator overloading and templates. The technique is referred to as 'lazy evaluation' or 'delayed evaluation' and the actual implementation details, which are fairly complex, may be found elsewhere [110].

Armadillo supports numerous standard linear algebra operations, as well as preliminary sparse matrix operation support; this allows us to make **mlpack** robust and able to handle both dense and sparse data with ease.

4.3 Template paradigms for fast, generic code

The primary template technique used in **mlpack** to provide robust, generic code is *policy-based design*, popularized by Andrei Alexandrescu in his book "Modern C++ Design" [109]. The central idea is that a class can be separated into orthogonal, modular components. One easy example is kernel principal components analysis [111], which has a clear parameter: the kernel. Given the ubiquity of object-oriented programming, most will see this as a perfect problem for inheritance, and construct a base `Kernel` class and derive all kinds of kernels from the base class, and then the `KernelPCA` class will take a downcasted

[3] Some obtuse implementations might store the result of the final multiplication in a temporary matrix before storing it in `Y`. I'd like to hope such an implementation was never released as 'production-quality code', but, having seen the code quality in the some of the internals of MATLAB, it is certainly possible...

base `Kernel` object and use that; this is exactly what the SHOGUN toolbox does [107]. There is a big problem, here, though: inheritance is not free (or more specifically, virtual inheritance and virtual methods in C++). Each call to a method of the base `Kernel` method incurs a pointer lookup to the derived kernel class.

For a method like kernel PCA, where a dataset of size N means the kernel must be evaluated $O(N^2)$ time, the overhead of this lookup is non-negligible. Instead of inheritance, we should use templates, which are actually more flexible. Consider the following `KernelPCA` class, with a template parameter for the kernel:

```
template<typename KernelType> class KernelPCA;
```

In this case, the type of the kernel is known at compile-time, and thus there is no need for a pointer lookup. If we assume (or say in the documentation) that each `KernelType` parameter will provide some `Evaluate()` function, we may easily evaluate the kernel inside the `KernelPCA` class by calling `KernelType::Evaluate()`.

This design pattern makes the `KernelPCA` class robust, generic, and extensible. Suppose a user wants to use some kernel which does not ship with the `KernelPCA` class. All they have to do is implement a class with an `Evaluate()` method, and they can plug their class in as the `KernelType` parameter; they do not need to know the internals of the `KernelPCA` class and in fact they don't even need to know how kernel PCA works, or consider any of the internal details of the class.

The overarching principle of policy-based design is that each component that is a template parameter has orthogonal functionality and no dependence on the other parameters. Thus, we can further generalize our kernel PCA class: some other policies might be a sampling strategy (i.e. `NystroemMethod`, `NoSampling`, `RandomSampling`, and so forth), element precision (`float`, `double`, etc.), and data matrix type (sparse or dense data matrices).

One major drawback of policy-based design is that there is not a clean way to enforce the interface requirements of a template parameter. In the context of our kernel PCA

example, there is no way in C++ to *require* that the `KernelType` type implements the `Evaluate()` method with the proper signature. Unfortunately, C++ Concepts, a solution to this, was not included in the C++11 standard; however, in some cases, SFINAE ("substitution failure is not an error") can be used to mitigate this issue.

4.4 Design principles of mlpack

As mentioned earlier, there are four overarching guidelines of **mlpack** development:

- to implement **scalable, fast** machine learning algorithms,
- to design an **intuitive, consistent, and simple** API for non-expert users,
- to implement a **variety** of machine learning methods, and
- to provide **cutting-edge** machine learning algorithms unavailable elsewhere.

We can consider each of these individually in turn.

4.4.1 Scalable and fast machine learning algorithms

Implementing scalable and fast machine learning algorithms often means that the simplest implementation of a technique will not suffice. Nearest neighbor search—a primary problem in this thesis and a primary component of **mlpack**'s applicability—thus is not implemented as a linear scan over all points; as we have already discussed, this is slow and does not scale. Instead, **mlpack** uses dual-tree algorithms to solve that problem, and numerous other problems where dual-tree algorithms are applicable. In addition, other techniques are also available, such as Nyström sampling for kernel PCA, and a low-rank semidefinite program solver [112].

But scalability and speed doesn't just correspond to implementing the asymptotically fastest algorithms. Numerous implementational tricks are often required, and where possible **mlpack** uses template techniques for speed. These techniques are both effective, and benchmarks from the **mlpack** paper show this [87].

Table 7: **mlpack** benchmark dataset sizes.

Dataset	**wine**	**cloud**	**wine-q**	**isolet**	**mboone**
UCI Name	Wine	Cloud	Wine Quality	ISOLET	MiniBooNE
Size	178x13	2048x10	6497x11	7797x617	130064x50

Dataset	**yp-msd**	**corel**	**covtype**	**mnist**	**randu**
UCI Name	YearPredictionMSD	Corel	Covertype	*N/A*	*N/A*
Size	515345x90	37749x32	581082x54	70000x784	1000000x10

Table 8: All-*k*-nearest neighbor benchmarks (in seconds).

Dataset	**mlpack**	Weka	Shogun	MATLAB	mlpy	`scikit`
wine	**0.0003**	0.0621	0.0277	0.0021	0.0025	0.0008
cloud	**0.0069**	0.1174	0.5000	0.0210	0.3520	0.0192
wine-q	**0.0290**	0.8868	4.3617	0.6465	4.0431	0.1668
isolet	**13.0197**	213.4735	37.6190	46.9518	52.0437	46.8016
mboone	**20.2045**	216.1469	2351.4637	1088.1127	3219.2696	714.2385
yp-msd	**5430.0478**	>9000.0000	>9000.0000	>9000.0000	>9000.0000	>9000.0000
corel	**4.9716**	14.4264	555.9600	60.8496	209.5056	160.4597
covtype	**14.3449**	45.9912	>9000.0000	>9000.0000	>9000.0000	651.6259
mnist	**2719.8087**	>9000.0000	3536.4477	4838.6747	5192.3586	5363.9650
randu	**1020.9142**	2665.0921	>9000.0000	1679.2893	>9000.0000	8780.0176

First, considering the task of nearest neighbor search, we compare **mlpack** against Weka [103], SHOGUN [107], MATLAB, mlpy [98], and `scikit-learn` [97][4]. **mlpack** contains a dual-tree algorithm for nearest neighbor search, and both `scikit-learn` and Weka contain a single-tree algorithm. MATLAB and SHOGUN and mlpy, however, contain the brute-force search. Using the datasets in Table 7, all-nearest-neighbor search is run for each dataset, and the results are presented in Table 8.

The next benchmark focuses on implementation efficiency, not algorithmic efficiency. We compare *k*-means for the same libraries, starting from the same centroids for each library (except mlpy and Weka, which did not allow specification of the starting centroids). Each library implements the standard *k*-means algorithm, which is a linear scan over every

[4]This comparison was performed in 2012, using **mlpack** 1.0.3 and period-appropriate versions of the other libraries. Things have changed since then—but **mlpack**'s automatic benchmarking system [113] shows that **mlpack** continues to hold a competitive edge for nearest neighbor search and other machine learning algorithms; see `http://www.mlpack.org/benchmarks.html`.

Table 9: *k*-means benchmarks (in seconds).

Dataset	Clusters	**mlpack**	Shogun	MATLAB	`scikit`
wine	3	**0.0006**	0.0073	0.0055	0.0064
cloud	5	**0.0036**	0.1240	0.0194	0.1753
wine-q	7	**0.0221**	0.6030	0.0987	4.0407
isolet	26	**4.9762**	8.5093	54.7463	7.0902
mboone	2	**0.1853**	8.0206	0.7221	*memory*
yp-msd	10	**34.8223**	135.8853	269.7302	*memory*
corel	10	**0.4672**	2.4237	1.6318	*memory*
covtype	7	**13.5997**	71.1283	54.9034	*memory*
mnist	10	**80.2092**	163.7513	133.9970	*memory*
randu	75	**727.1498**	7443.2675	3117.5177	*memory*

point and every cluster to find the nearest cluster centroid of every point (later, we will describe a fast dual-tree algorithm for this); thus, each library is doing the same amount of work. Still, we see that **mlpack**'s implementation is superior in efficiency to the other libraries, in Table 9.

4.4.2 Intuitive, consistent, and simple API

The second focus of **mlpack** development is consistent, easy-to-use code. To this end, **mlpack** is organized into two large directories of code: `core/` and `methods/`. The code in `core/` includes things like probability distributions, metrics, kernels, and other utility functionality that makes up the building blocks of the machine learning methods implemented in `methods/`.

In general, classes in **mlpack** operate in the same way: the constructor performs preprocessing and *must* return a valid object which is ready to be trained or used. Training (for a machine learning method) then must take place in an `Estimate()` or `Train()` method. Then, application of the machine learning model to data should be through a function such as `Cluster()`, `Predict()`, `Regress()`, or a similarly descriptive-named function.

Further, classes in **mlpack** implement policy-based design in the manner discussed in Section 4.3, and each template parameter (when possible) should have a default to improve usability. This means that virtually every class in **mlpack** is extensible and modular, so that

users who are not familiar with the internals of the codebase can still implement their own modifications of the algorithms that **mlpack** provides. Below is an example interface, for *k*-means clustering:

```
template<typename MetricType = metric::EuclideanDistance,
         typename InitialPartitionPolicy = RandomPartition,
         typename EmptyClusterPolicy = MaxVarianceNewCluster,
         template<class, class> class LloydStepType = NaiveKMeans,
         typename MatType = arma::mat>
class KMeans;
```

In this example, the *k*-means implementation provides five significant degrees of freedom to the user. First, the user may choose their own distance metric[5]; but, if they choose not to, the standard L2 distance is the default. The second template parameter allows the user to specify the way that initial clusters are assigned; the third allows the user to specify the action to be taken when an empty cluster (that is, a cluster that owns no points) is detected at the end of an iteration. The `LloydStepType`, a *template template parameter*[6], allows the user to specify their own algorithm to perform a single *k*-means iteration. The default is the brute-force implementation used for the benchmarks, but **mlpack** provides four other techniques that a user can plug in, too. Lastly, the user may specify the type of data: dense (`arma::mat`) or sparse (`arma::sp_mat`).

Lastly, not every person who wishes to use machine learning software is familiar with C++; therefore, each machine learning algorithm implemented by **mlpack** also includes a well-documented command-line interface. This follows the UNIX tradition of providing small tools which can be wrapped together into larger applications. To continue with the *k*-means example from just above, below demonstrates how *k*-means can be run from the command line once **mlpack** is built and installed:

```
$ kmeans -i dataset.csv -c 25 -a elkan -v -C centroids.csv
```

[5] In our implementation, this is limited to Euclidean metrics, but that is a minor detail that the documentation clarifies.

[6] C++ is horribly confusing, and with terms like 'template template parameter' and 'rvalue reference' and 'functor' and 'partial template function specialization', the entire language is quite possibly an advanced form of satire. Still, the generic programming infrastructure is quite powerful, as the text shows; personally, I consider C++ both the worst and the best language ever designed.

The example above performs *k*-means clustering with 25 clusters on the dataset in `dataset.csv`, using Elkan's algorithm, and storing the converged centroids in a file named `centroids.csv`. These are only a few of the options available, though; the rest can be accessed with either `man kmeans` or `kmeans -h`. This pattern is consistent across the rest of the **mlpack** methods.

Although this discussion has been relatively short, it is still possible to see how a robust, modular design can lead to flexibility for advanced users while maintaining simplicity for novice users.

4.4.3 Current functionality of mlpack

The last two goals of **mlpack**, a variety of methods and cutting-edge algorithms unavailable elsewhere, are represented easily by a list of the current functionality of **mlpack**. Given below are each of the significant modules that **mlpack** provides, with relevant citations. A **(*)** indicates that **mlpack** has the only implementation of the technique.

- Dual-tree *k*-nearest neighbor search **(*)** [66]
- Dual-tree *k*-furthest neighbor search **(*)**
- Dual-tree range search **(*)**
- Dual-tree Borůvka's algorithm for minimum spanning tree computation **(*)** [68]
- Dual-tree fast max-kernel search **(*)** [46, 79]
- Rank-approximate nearest neighbor search **(*)** [67]
- Adaboost [114]
- Non-negative matrix factorization [115]
- SVD batch learning matrix factorization [116]
- SVD incremental learning matrix factorization [116]

- Least-squares linear and ridge regression
- Naive Bayes classifier
- Least angle regression [117]
- Gaussian mixture model training and prediction
- Density estimation trees [118]
- Mean shift clustering [119]
- Locality-sensitive hashing based on p-stable distributions [120]
- Neighborhood components analysis [121]
- Local coordinate coding [122]
- Sparse coding via LARS [123]
- Dual-tree k-means clustering **(*)** plus four other k-means techniques [80]
- Kernel principal components analysis [111]
- QUIC-SVD **(*)** [58]
- Perceptrons [11]
- Low-rank semidefinite program solver [112]
- General artificial neural network framework
- Hidden Markov model training, prediction, and generation
- Principal components analysis
- Softmax regression
- Collaborative filtering via matrix decomposition [124]

- Nyström method for sampling [24]

4.5 Tree-independent dual-tree algorithms in mlpack

The current chapter, up to this point, has been something of a deviation from the rest of this thesis; it has been focused on a general-purpose machine learning toolkit. However, one part of **mlpack** is of particular importance to this thesis, and builds upon the material just introduced: the dual-tree algorithm framework in **mlpack**.

We already know from Chapter 3 that dual-tree algorithms may be represented by three separate components: a type of tree, a type of traversal, and a `Score()` and `BaseCase()` function. This allows us to construct a relatively straightforward API for dual-tree algorithms. The API reference given here is as of **mlpack** 1.1.0[7].

In our API, we assume that a tree is built on a dataset of type `MatType` (usually `arma::mat`), and each point in this dataset is indexable by a unique unsigned integer type, `size_t`. A point in the input space, when not represented by an index, is represented by a `VecType` object (usually `arma::vec`). The type of the dataset will follow the same API conventions as Armadillo matrices; however, in this discussion, we don't need to dig that deep.

4.5.1 The `TreeType` policy

The first API to establish is the `TreeType` class policy, which defines the methods that a tree type must implement. In **mlpack**, there is no distinction between a *tree* and a *node*, because each node corresponds to its own subtree. Therefore, only one class is necessary to represent a tree and all nodes contained in that tree. The class must implement each of the methods required by the `TreeType` class policy. Figure 14 and 15 show most of the methods that a tree type must implement. There are a few others not documented here, but the excerpt given here is sufficient to get a feel for the API. The comments above each method describe the required functionality.

[7] Yet to be released—but the API is finalized.

```
// Return the dataset that the tree is built on.
const MatType& Dataset();

// Return the parent of the node, or NULL if this is the root.
TreeType* Parent();

// Return the number of children held by the node.
size_t NumChildren();

// Return the ith child held by the node.
TreeType& Child(const size_t i);

// Return the number of points held by the node.
size_t NumPoints();

// Return the ith point held by the node.
VecType& Point(const size_t i);

// Return the number of descendant points of the node.
size_t NumDescendants();

// Return the ith descendant point of the node.
VecType& Descendant(const size_t i);

// Return the number of descendant nodes.
size_t NumDescendantNodes();

// Return the ith descendant node.
VecType& DescendantNode(const size_t i);

// Store the centroid of the node in the given vector.
void Centroid(VecType& centroid);
```

Figure 14: Methods required by `TreeType` class (part one).

Most of the methods have an analog to quantities or bounds described in Chapter 3 (specifically, Section 3.3). Points, children, and descendant points are all accessible through the API. It is important to note that points are referred to by their index in the dataset (of unsigned integer type `size_t`). Methods such as `MinDistance()` and `MaxDistance()` correspond to the bounds $d_{\min}(\cdot, \cdot)$ and $d_{\max}(\cdot, \cdot)$, respectively. Each of the methods in this API are directly usable and useful by the soon-to-be-formalized-in-code `BaseCase()` and `Score()` functions.

However, it is often the case that the API provided by the `TreeType` policy is insufficient for certain dual-tree algorithms. For instance, suppose the existence of a dual-tree algorithm which requires on the kurtosis of the descendant points of each node. For this

```
// Get the distance between the centroid of this node and the centroid
// of its parent.
double ParentDistance();

// Return the furthest distance from the centroid of the node to any
// point held in the node.
double FurthestPointDistance();

// Return the furthest descendant point distance from the centroid
// of the node.
double FurthestDescendantDistance();

// Return the minimum distance between the given point and the node.
template<typename OtherVecType>
double MinDistance(OtherVecType& point);

// Return the minimum distance between the given node and this node.
double MinDistance(TreeType& otherNode);

// Return the maximum distance between the given point and the node.
template<typename OtherVecType>
double MaxDistance(OtherVecType& point);

// Return the maximum distance between the given node and this node.
double MaxDistance(TreeType& otherNode);
```

Figure 15: Methods required by `TreeType` class (part two).

niche situation, it is unreasonable to add a `Kurtosis()` function to the `TreeType` policy; acquiescing to each of these requests quickly leads to a wildly complex set of requirements for tree implementers. The solution for **mlpack** is simpler and more elegant: each class implementing the `TreeType` policy must have as a template parameter a `StatisticType` class, and must hold an instance of the `StatisticType` class in each node.

The `StatisticType` class (or just *the statistic*) is useful for holding problem-specific quantities such as the kurtosis, as in the above example, or other cached statistics similar to those suggested by Moore in the Anchors hierarchy [125]. The restrictions on the `StatisticType` API are very loose; a `StatisticType` must simply provide a default constructor, and a constructor called with the node that the statistic corresponds to. An example class definition is given for the kurtosis example in Figure 16.

In general, most dual-tree algorithms in **mlpack** implement their own statistic type to cache bounding information, centroids, centers, or other quantities not provided for by the

```
class KurtosisStatistic
{
 public:
  // Default constructor; sets kurtosis to 0.
  KurtosisStatistic();

  // Construct the KurtosisStatistic on the given node.
  // This calculates the kurtosis on the descendant points of the node.
  KurtosisStatistic(TreeType& node);

  // More methods and members may (and should) be added, but the two
  // above are all that is required.
};
```

Figure 16: Example `StatisticType` class.

general `TreeType` class policy.

There is one more important part of the **mlpack** `TreeType` class: the `TreeTraits` class for template metaprogramming. We may use the `TreeTraits` class to determine traits about the tree at compile-time (similar to the traits in Tables 2, 3, 4, 5, and 6), which may allow us to make additional assumptions in the `BaseCase()` and `Rules()` function. We must simply define an appropriate specialization of the `TreeTraits` template class. An example, for the cover tree, is given in Figure 17; note that in this example, the specialization itself must be templatized because the `CoverTree` class is templatized too (in accordance with the principles of policy-based design).

Using these traits in practice is fairly straightforward. Consider the simple but common task of recursing into the children of a node[8]. This can be written generally using the API we have introduced (in Figure 18):

But we can use `TreeTraits<TreeType>::BinaryTree` to avoid the loop entirely for binary trees (Figure 19). The `if` statement is known at compile-time, and therefore the compiler can optimize away the `for` loop entirely if the tree is a binary tree.

This strategy of using compile-time constants to allow the compiler to use specialized

[8]This example is somewhat contrived. Most modern optimizing compilers would be able to unroll the loop completely for binary trees, without the hint provided by template metaprogramming. However, the example is still useful in that it demonstrates how one might use the `TreeTraits<>` class in more complex situations that the compiler will not optimize correctly.

```
template<typename MetricType,
         typename RootPointPolicy,
         typename StatisticType,
         typename MatType>
class TreeTraits<CoverTree<MetricType, RootPointPolicy,
                           StatisticType, MatType>>
{
 public:
  /**
   * The cover tree (or, this implementation of it) does not require
   * that children represent non-overlapping subsets of the parent
   * node.
   */
  static const bool HasOverlappingChildren = true;

  /**
   * Each cover tree node contains only one point, and that point is
   * its centroid.
   */
  static const bool FirstPointIsCentroid = true;

  /**
   * Cover trees do have self-children.
   */
  static const bool HasSelfChildren = true;

  /**
   * Points are not rearranged when the tree is built.
   */
  static const bool RearrangesDataset = false;

  /**
   * The cover tree is not necessarily a binary tree.
   */
  static const bool BinaryTree = false;
};
```

Figure 17: Example specialization of `TreeTraits`.

sections of code is applicable to far more complex situations than the one described above, and is used at length in **mlpack** to accelerate algorithms.

```
for(size_t i = 0; i < node.NumChildren(); ++i)
  Recurse(node.Child(i));
```

Figure 18: Recursing into the children of a node.

```
if (TreeTraits<TreeType>::BinaryTree)
{
  Recurse(node.Child(0));
  Recurse(node.Child(1));
}
else
{
  for(size_t i = 0; i < node.NumChildren(); ++i)
    Recurse(node.Child(i));
}
```

Figure 19: Compile-time specialization of recursion with `TreeTraits`.

```
// Compute the base case between a query and reference point.
double BaseCase(const size_t queryIndex, size_t referenceIndex);

// A single-tree Score() function; returns DBL_MAX if the node should
// be pruned.
double Score(const size_t queryIndex, TreeType& referenceNode);

// Re-score the node, considering the old score.
double Rescore(const size_t queryIndex,
               TreeType& referenceNode,
               const double oldScore);

// A dual-tree Score() function; returns DBL_MAX if the combination
// should be pruned.
double Score(TreeType& queryNode, TreeType& referenceNode);

// Re-score the node combination, using the old score.
double Rescore(TreeType& queryNode,
               TreeType& referenceNode,
               const double oldScore);
```

Figure 20: Required API for `RuleType` classes.

4.5.2 The `RuleType` policy

The next part of the dual-tree algorithm infrastructure in **mlpack** is the `RuleType` policy class, which encapsulates the `BaseCase()` and `Score()` functions that describe the dual-tree (or single-tree) algorithm. These classes end up being more complex than the paper-presented abstraction would suggest, with the necessary methods being listed in Figure 20. Short comments for each method are given in the figure; details are given below.

The `BaseCase()` and `Score()` methods are nearly as one would expect. `BaseCase()` takes a query point and reference point, but returns a `double`, somewhat contrary to the abstraction. The reason for this is that sometimes, the result of the `BaseCase()` calculation is useful elsewhere. Consider nearest-neighbor search with ball trees. The base case calculates the distance between a query point and a reference point, but this is also usable during `Score()`, if the query and reference points p_q and p_r are the centers of tree nodes $\mathscr{N}_q$ and $\mathscr{N}_r$. Inside the `Score()` method, one can call `BaseCase()` directly, which will return $d(p_q, p_r)$. This may be easily adjusted into $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$, by subtracting the radii of the two nodes (λ_q and λ_r). Some semi-clever software engineering is then necessary to make sure the subsequent calls to `BaseCase()` with p_q and p_r (which are required by the definition of pruning dual-tree traversal) don't perform any work. For certain types of trees, a significant amount of computation can be avoided in this manner. However, because this is entirely implementation-specific, there is no need to discuss this detail in the abstract terminology of the previous chapter.

Because ∞ is difficult to represent in C++[9], the `Score()` functions return `DBL_MAX` when pruning is possible instead of ∞. Any value less than `DBL_MAX` indicates a score for prioritized recursion, with lower scores indicating that the node combination should be recursed into sooner.

The last deviation from the abstraction is the two `Rescore()` functions. The pruning rules for most dual-tree algorithms boil down to some comparison of the form "if a is greater than b, then prune", where a is some combination-specific quantity that usually relates to the distance between the nodes, and b is some bound quantity. For instance, in single-tree nearest neighbor search, a is $d_{\min}(p_q, \mathscr{N}_r)$ and b is $d(p_q, \hat{p}_{nn})$, where $\hat{p}_{nn}$ is the current nearest neighbor candidate for the query point p_q. In these situations, a (or some algebraic manipulation thereof) is often a good score to return to the traversal for prioritized recursion. The `Rescore()` function allows the traversal to make a final check for pruning

[9]More specifically, it takes longer and is uglier to type `std::numeric_limits<double>::infty()` than `DBL_MAX`.

```
template<typename RuleType>
class ExampleTraversal
{
 public:
  // Construct the traversal with the given instantiated RuleType.
  ExampleTraversal(RuleType& rule);

  // Perform a single-tree traversal on the given query point and
  // reference tree.
  template<typename TreeType>
  void Traverse(const size_t queryIndex, TreeType& referenceNode);

  // Perform a dual-tree traversal on the given query and reference
  // tree.
  template<typename TreeType>
  void Traverse(TreeType& queryNode, TreeType& referenceNode);

  // More methods are allowable, but not required.
};
```

Figure 21: Example `TraversalType` class.

before recursion, in case the bounding quantity b has tightened between the call to `Score()` and the recursion.

It is important to note that the `Rescore()` functions are not required, and in situations where it is known that `Rescore()` cannot be effective, the body of the `Rescore()` function can simply be '`return oldScore;`'.

4.5.3 The `TraversalType` policy

The last piece of **mlpack**'s dual-tree algorithm puzzle is the `TraversalType` policy, which uses both the `RuleType` and `TreeType` policies. The requirements are quite simple and are shown in Figure 21. The traversal must take a `RuleType` class as its only template parameter, have a constructor which takes an instantiated `RuleType`, and provide a `Traverse()` function to perform either a single-tree or dual-tree traversal (or a `Traverse()` function for each case may be provided, like in the example). The `Traverse()` overload for the single-tree case takes a query point (`size_t`) and a reference node (`TreeType&`); in the dual-tree case, `Traverse()` takes a query node and reference node, both of type `TreeType&`. For either overload, the node is *not* required to be the root of a tree, which means that a traversal

```
// Assume we have datasets from somewhere.
extern arma::vec queryPoint;
extern arma::mat queryData, referenceData;

// Build the trees.
MyTree queryTree(queryData);
MyTree referenceTree(referenceData);

// Create the rule.  We assume MyRule has a default constructor (this
// assumption is generally not true, though, so the code ends up being
// slightly more complex).
MyRule rule;

// Create the traversal.
MyTraversal traversal(rule);

// First run a single-tree traversal with the query point.
traversal.Traverse(queryPoint, referenceTree);

// Now run a dual-tree traversal with the query tree.
traversal.Traverse(queryTree, referenceTree);
```

Figure 22: Code to run a sample dual-tree algorithm in **mlpack**.

can be run only on subtrees, if so desired.

This API leaves a lot of flexibility to the implementer; additional functions may be provided if desired. Further, `Traverse()` is not required to be recursive; it is only the starting point for a traversal.

4.5.4 Assembling a dual-tree algorithm in mlpack

With the `TreeType`, `RuleType`, and `TraversalType` policy classes, which represent each of the pieces of a dual-tree algorithm, it is easy to assemble a dual-tree (or single-tree) algorithm. Figure 22 shows some example code that assembles a dual-tree algorithm using `MyTree` as the tree type, `MyRule` as the rule type, and `MyTraversal` as the traversal type.

In reality, the dual-tree algorithms implemented in **mlpack** tend to be quite more complex than the snippet given. In particular, `RuleType` classes often have constructors that take a number of matrices to store the results of the dual-tree algorithm, and sometimes other tuning parameters or options. Both traversals and trees may also have options that can be (optionally) set in the constructor.

Nonetheless, the dual-tree algorithm API as detailed in this section is sufficient to encapsulate the class of dual-tree algorithms.

Figure 23: Fritz and Drusilla.[10]

(a) Fritz.

(b) Drusilla.

[10] I was originally challenged to include a picture of my cats somewhere in this document. In an unusual moment of lucid reasoning, I decided that it would not be a good idea. However, during discussions with the committee, the point came up (as a joke) and the committee seemed to be of the opinion that inclusion of the cats would not detract from the thesis, especially given that McClellan [126] did so much more obviously on the front cover of his book. Therefore, I have finally published an academic document with a picture of my cats in it, fulfilling a long-standing bet. I believe that I am now owed lunch, or a beer, or something.

CHAPTER 5

TREES

The first piece of a dual-tree algorithm is the type of space tree. As with each piece of the tree-independent dual-tree algorithm abstraction detailed in Chapter 3, any improvements to trees can propagate to the algorithms that use that particular type of tree. Therefore, I have spent some time working with individual tree types, and the advancements and other insights I have found are thus detailed in this chapter. The majority of my work—and all of the work worth publishing in this chapter—concerns the cover tree, a complex tree type that is most useful for its theoretical properties.

5.1 Free parameters in the cover tree

The cover tree is a widely-known tree structure that has gained popularity for nearest neighbor search and other tasks such as local SVM training [127], max-kernel search [46, 79], Euclidean minimum spanning tree calculation [68], and k-average-medoid calculation [128] (as well as many others). A large part of the interest in the structure comes from its convenient theoretical properties, which allow bounding the complexity of the structure with respect to a dataset-dependent quantity called the *expansion constant*.

However, the cover tree is an extremely complex structure and has been found repeatedly to have a lot of tree-building overhead [129, 130]; the implementation in **mlpack** shows this assessment to be accurate, with cover tree construction often taking significantly longer than *kd*-tree construction (or other tree types).

The cover tree construction algorithm, as given, has several parameters for tuning, including the selection of the root of the tree, which I therefore investigated. I also attempted to find those characteristics of the cover tree which correlate to better performance. Unfortunately, the only correlation I have found to date is a weak correlation, and my efforts to select a root point better were a wash. Still, I find the negative results to be valuable, if only

as a warning to others, and therefore we will discuss them (briefly).

5.1.1 The cover tree: a rehash

The cover tree is a leveled hierarchical data structure originally proposed for the task of nearest neighbor search by Beygelzimer, Kakade, and Langford [57]. Each node $\mathcal{N}_i$ in the cover tree is associated with a single point p_i. An adequate description is given in their work (we have adapted notation slightly):

> A *cover tree* $\mathcal{T}$ on a dataset S is a leveled tree where each level is a "cover" for the level beneath it. Each level is indexed by an integer scale s_i which decreases as the tree is descended. Every *node* in the tree is associated with a point in S. Each *point* in S may be associated with multiple nodes in the tree; however, we require that any point appears at most once in every level. Let C_{s_i} denote the set of points in S associated with the nodes at level s_i. The cover tree obeys the following invariants for all s_i:
>
> - *(Nesting).* $C_{s_i} \subset C_{s_i-1}$. This implies that once a point $p \in S$ appears in C_{s_i} then *every* lower level in the tree has a node associated with p.
> - *(Covering tree).* For every $p_i \in C_{s_i-1}$, there exists a $p_j \in C_{s_i}$ such that $d(p_i, p_j) < 2^{s_i}$ and the node in level s_i associated with p_j is a parent of the node in level $s_i - 1$ associated with p_i.
> - *(Separation).* For all distinct $p_i, p_j \in C_{s_i}$, $d(p_i, p_j) > 2^{s_i}$.

As a consequence of this definition, if there exists a node $\mathcal{N}_i$, containing the point p_i at some scale s_i, then there will also exist a self-child node $\mathcal{N}_{ic}$ containing the point p_i at scale $s_i - 1$ which is a child of $\mathcal{N}_i$. In addition, every descendant point of the node $\mathcal{N}_i$ is contained within a ball of radius 2^{s_i+1} centered at the point p_i; therefore, $\lambda_i = 2^{s_i+1}$ and $\mu_i = p_i$ (see Table 1).

Note that the cover tree may be interpreted as an infinite-leveled tree, with the highest level C_∞ containing only the root point, the lowest level $C_{-\infty}$ containing every point (that is, $C_{-\infty} = S$), and all levels between defined as above. Beygelzimer et al. find this representation (which they call the *implicit* representation) easier for description of their algorithms and some of their proofs [57]. But clearly, this is not suitable for implementation; hence, they also introduce an *explicit* representation in which all nodes that have only a self-child are coalesced upwards (that is, the node's self-child is removed, and the children of that self-child are taken to be the children of the node).

The theoretical results for the cover tree apply to both the explicit and implicit representations of the tree. Personally, I think that the explicit representation is much simpler to work with; therefore, all of the following results work with the explicit representation of the tree. Some key points of the explicit representation of a cover tree are listed below (see also Table 6 and Section 5.1.1):

- Each node $\mathscr{N}_i$ contains a single point p_i and has a scale s_i.
- All descendant points of $\mathscr{N}_i$ are within the radius 2^{s_i+1}.
- The children of $\mathscr{N}_i$ are *not* necessarily at the scale $s_i - 1$, but their scale must be less than s_i.
- The regions corresponded to by nodes $\mathscr{N}_i$ and $\mathscr{N}_j$ where $s_i = s_j$ may overlap ($\mathscr{N}_i$ corresponds to a ball of radius 2^{s_i+1} and center p_i; $\mathscr{N}_j$ corresponds to a ball of the same radius and center p_j), but the nodes are still beholden to the separation invariant.

We are not concerned with a batch construction algorithm for the cover tree here, but the interested reader should refer to the original paper for details [57]. The tree can be built in $O(c^6 N \log N)$ time, where c is the expansion constant (described in the following subsection).

5.1.2 The expansion constant

The explicit representation of a cover tree has a number of useful theoretical properties based on the expansion constant [131]; we restate its definition below.

Definition 8. *Let $B_S(p, \Delta)$ be the set of points in S within a closed ball of radius Δ around some $p \in S$ with respect to a metric d: $B_S(p, \Delta) = \{r \in S : d(p, r) \leq \Delta\}$. Then, the* **expansion constant** *of S with respect to the metric d is the smallest $c \geq 2$ such that*

$$|B_S(p, 2\Delta)| \leq c|B_S(p, \Delta)| \; \forall \; p \in S, \; \forall \; \Delta > 0. \tag{8}$$

The expansion constant is used heavily in the cover tree literature. It is, in some sense, a notion of instrinic dimensionality, and previous work has shown that there are many scenarios where c is independent of the number of points in the dataset [131, 57, 132, 133]. Note also that if points in $S \subset \mathcal{H}$ are being drawn according to a stationary distribution $f(x)$, then c will converge to some finite value c_f as $|S| \to \infty$. To see this, define c_f as a generalization of the expansion constant for distributions. $c_f \geq 2$ is the smallest value such that

$$\left(\int_{\mathcal{B}_{\mathcal{H}}(p, 2\Delta)} f(x)dx \right) \leq c_f \left(\int_{\mathcal{B}_{\mathcal{H}}(p, \Delta)} f(x)dx \right) \tag{9}$$

for all $p \in \mathcal{H}$ and $\Delta > 0$ such that $\int_{\mathcal{B}_{\mathcal{H}}(p, \Delta)} f(x)dx > 0$, and with $\mathcal{B}_{\mathcal{H}}(p, \Delta)$ defined as the closed ball of radius Δ in the space $\mathcal{H}$.

As a simple example, take $f(x)$ as a uniform spherical distribution in $\mathcal{R}^d$: for any $|x| \leq 1$, $f(x)$ is a constant; for $|x| > 1$, $f(x) = 0$. It is easy to see that c_f in this situation is 2^d, and thus for some dataset S, c must converge to that value as more and more points are added to S. Closed-form solutions for c_f for more complex distributions are less easy to derive; however, empirical speedup results from the original cover tree paper suggest the existence of datasets where c is not strongly dependent on d [57]. For instance, the `covertype` dataset has 54 dimensions but the expansion constant is much smaller than other, lower-dimensional datasets.

There are some other important observations about the behavior of c. Adding a single point to S may increase c arbitrarily: consider a set S distributed entirely on the surface of a unit hypersphere. If one adds a single point at the origin, producing the set S', then c explodes to $|S'|$ whereas before it may have been much smaller than $|S|$. Adding a single point may also decrease c significantly. Suppose one adds a point arbitrarily close to the origin to S'; now, the expansion constant will be $|S'|/2$. Both of these situations are degenerate cases not commonly encountered in real-world behavior; we discuss them in order to point out that although we can bound the behavior of c as $|S| \to \infty$ for S from a stationary distribution, we are not able to easily say much about its convergence behavior.

The expansion constant can be used to show a few useful bounds on various properties of the cover tree; we restate these results below, given some cover tree built on a dataset S with expansion constant c and $|S| = N$:

- **Width bound:** no cover tree node has more than c^4 children (Lemma 4.1, [57]).
- **Depth bound:** the maximum depth of any node is $O(c^2 \log N)$ (Lemma 4.3, [57]).
- **Space bound:** a cover tree has $O(N)$ nodes (Theorem 1, [57]).

Lastly, we introduce a convenience lemma of our own which is a generalization of the packing arguments used by [57]. This is a more flexible version of their argument.

Lemma 1. *Consider a dataset S with expansion constant c and a subset $C \subseteq S$ such that every point in C is separated by δ. Then, for any point p (which may or may not be in S), and any radius $\rho\delta > 0$:*

$$|B_S(p, \rho\delta) \cap C| \leq c^{2+\lceil \log_2 \rho \rceil}. \tag{10}$$

Proof. The proof is based on the packing argument from Lemma 4.1 in [57]. Consider two cases: first, let $d(p, p_i) > \rho\delta$ for any $p_i \in S$. In this case, $B_S(p, \rho\delta) = \emptyset$ and the lemma

holds trivially. Otherwise, let $p_i \in S$ be a point such that $d(p, p_i) \leq \rho\delta$. Observe that $B_S(p, \rho\delta) \subseteq B_S(p_i, 2\rho\delta)$. Also, $|B_S(p_i, 2\rho\delta)| = c^{2+\lceil \log_2 \rho \rceil} |B_S(p_i, \delta/2)|$ by the definition of the expansion constant. Because each point in C is separated by δ, the number of points in $B_S(p, \rho\delta) \cap C$ is bounded by the number of disjoint balls of radius $\delta/2$ that can be packed into $B_S(p, \rho\delta)$. In the worst case, this packing is perfect, and

$$|B_S(p, \rho\delta)| \leq \frac{|B_S(p_i, 2\rho\delta)|}{|B_S(p_i, \delta/2)|} \leq c^{2+\lceil \log_2 \rho \rceil}. \tag{11}$$

□

5.1.3 Root point selection policy

Undeterred by the hearsay of others, who reported that they were unable to get any noticeable improvement out of the cover tree by tweaking build-time heuristics, I observed that one free parameter in the build process of the cover tree is the selection of the root point. The reference implementation simply selects the first point in the dataset (and **mlpack** does this too).

It is intuitive to reason that a good cover tree will have a root point near the centroid, which will allow the scale of the root node to potentially be smaller and may result in a more balanced tree. The centroid can be found quite quickly in a single pass, and then the nearest point in the dataset can be found by simply using nearest neighbor search on the dataset with the centroid as a query point. Although this is time-consuming, a simple heuristic may exist to find a point near enough to the centroid to still obtain the runtime benefits of selecting a root point near the centroid—if such runtime benefits existed.

Table 10 shows statistics for the search-time performance of trees built using standard construction techniques and the nearest-point-to-centroid root selection construction technique at build-time, and Table 11 shows build-time performance statistics on a variety of datasets.

More trials show no clear trends between the choice of root point and the performance

Table 10: Runtime statistics for different root point policies.

Dataset	Size ($n \times d$)	Root Policy	Base cases	Scores
cloud	10x2048	standard	100,294	200,777
cloud	10x2048	centroid	99,287	201,118
sat-train	37x4435	standard	796,257	1,453,644
sat-train	37x4435	centroid	774,093	1,417,471
isolet	617x7797	standard	36,151,613	50,368,497
isolet	617x7797	centroid	35,789,986	49,869,594
corel	32x37749	standard	96,773,720	206,686,912
corel	32x37749	centroid	100,138,961	210,169,714
miniboone	50x130064	standard	303,840,668	670,128,641
miniboone	50x130064	centroid	305,535,492	674,806,902
covertype	54x581012	standard	125,172,202	280,307,963
covertype	54x581012	centroid	125,052,402	279,764,213
mnist	784x70000	standard	2,545,638,649	3,750,433,910
mnist	784x70000	centroid	2,546,585,630	3,758,033,756

Table 11: Build-time statistics for different root point policies.

Dataset	Root Policy	Distance Evals	Nodes	max $\lvert\mathscr{C}_i\rvert$	$s_{\max}$	$s_{\min}$
cloud	standard	88,796	2,856	21	12	-1
cloud	centroid	91,540	2,871	22	12	-1
sat-train	standard	1,814,451	5,157	273	9	4
sat-train	centroid	1,752,514	5,155	295	9	4
isolet	standard	22,951,726	9,158	2112	5	2
isolet	centroid	22,495,666	9,147	2504	5	2
corel	standard	80,757,262	46,885	488	1	-7
corel	centroid	85,601,844	47,020	407	1	-7
miniboone	standard	239,336,248	160,348	626	24	4
miniboone	centroid	241,277,485	160,419	550	24	4
covertype	standard	160,801,654	801,884	71	14	2
covertype	centroid	164,858,576	801,864	98	14	2
mnist	standard	1,512,062,473	77,851	5410	12	8
mnist	centroid	1,468,179,161	77,882	5113	13	8

of the tree at runtime. Sometimes, the standard policy performs better, and sometimes, the centroid policy performs better. Swings of up to about 10% (in terms of number of base case computations) are observed. In those files, there is also data for when the base is set to 1.3, as in the original implementation [57]; it becomes clear that the speedup due to the smaller base is not due to a better tree—the number of base case computations almost

always goes up—but instead that the number of distance computations during tree-building can be orders of magnitude fewer. Often, with cover trees, the longest part of the search is the tree-building, due to the complexity of the cover tree building procedure.

5.1.4 Correlation of tree width to performance

To further investigate the effect of root point selection, on a few sample datasets ('cloud', 'sat-train', and 'winequality', all from the UCI repository [134]), cover trees were made

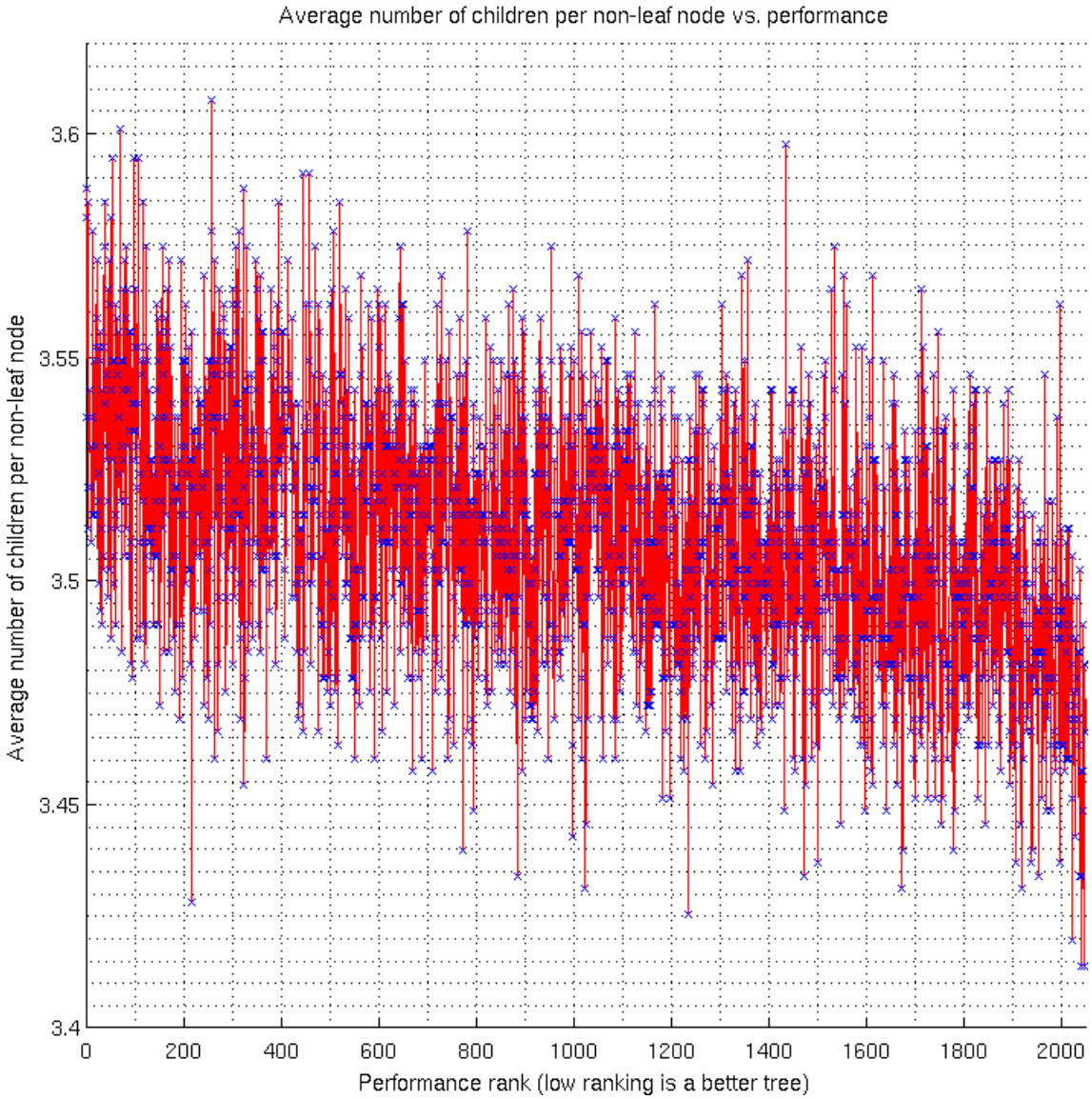


Figure 24: Tree performance related to the average number of children per node, for the cloud dataset.

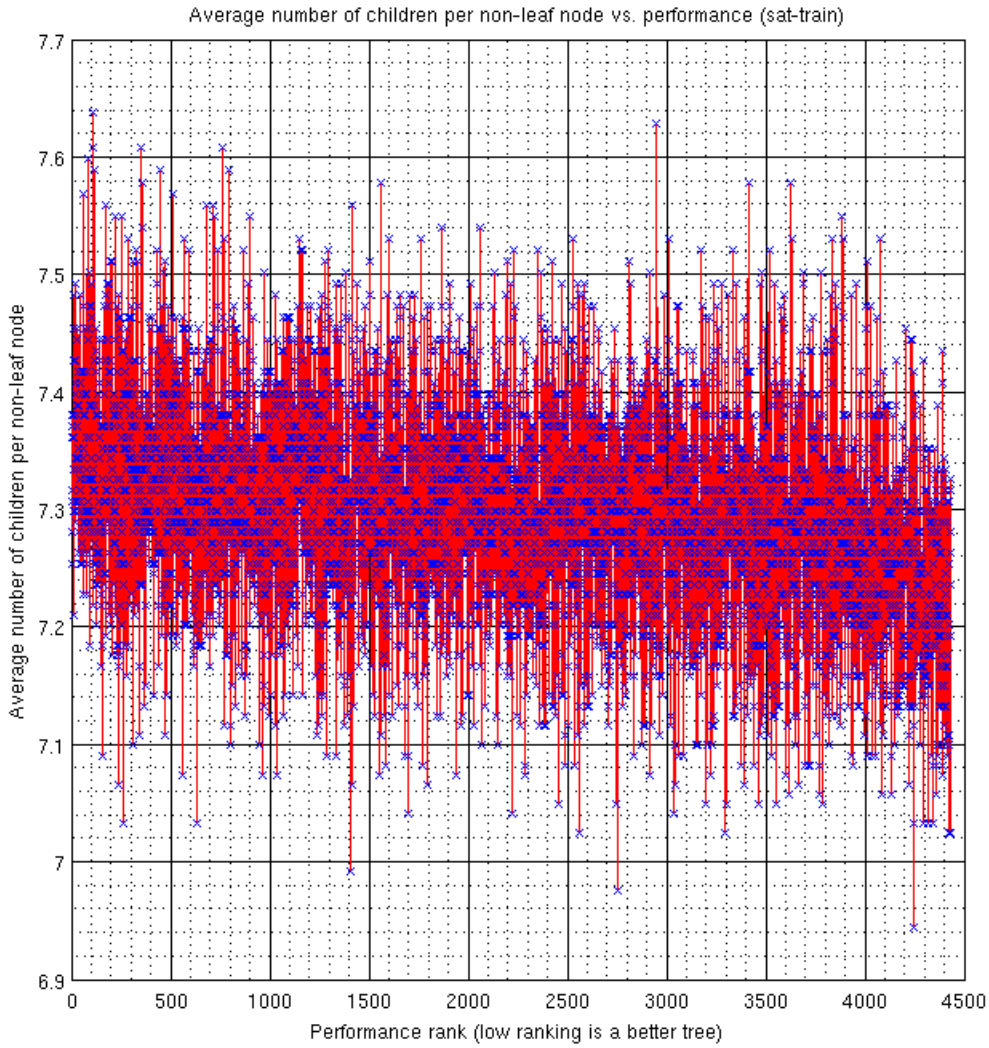


Figure 25: Tree performance related to the average number of children per node, for the sat-train dataset.

with every possible root point. Then, each tree was used to perform dual-tree all-nearest-neighbor search. Overall, the difference between the number of distance calculations performed during search for the best and worst trees was less than 20% for each dataset, meaning that even if there was a good way to select the root point, it would not affect performance significantly.

The only correlation found between these trees and their performance was the average number of children per node, and even that was only a weak correlation. Figures 24, 25, and

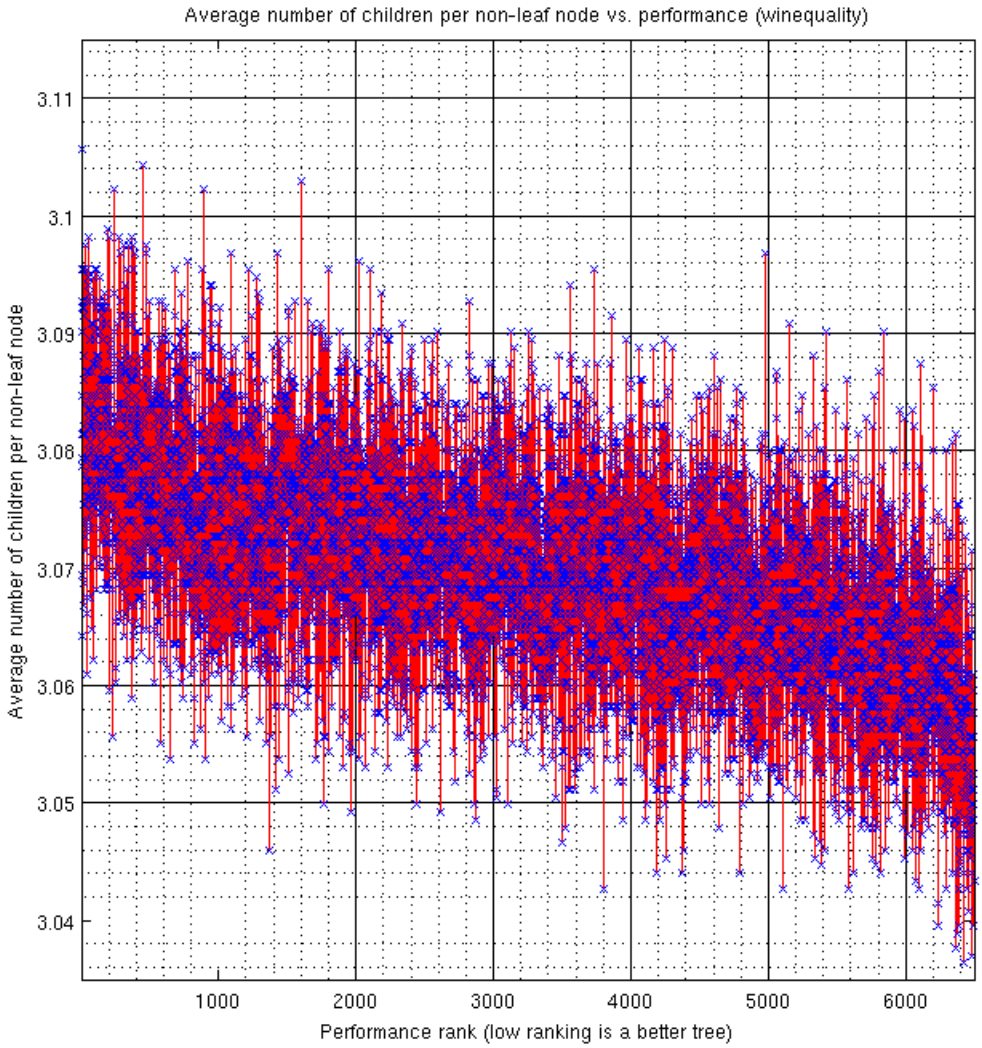


Figure 26: Tree performance related to the average number of children per node, for the winequality dataset.

26 show this relation for the ‘cloud’, ‘sat-train’, and ‘winequality’ datasets, by sorting the trees by their performance (with better-performing trees coming before worse-performing trees), and then generating a scatter plot using the average number of children per node.

Intuitively, the reason that trees with lower average numbers of children perform worse is probably because those trees are more likely to have very unbalanced pectinate branches; that is, branches where each node only has two children, and one of those children is a leaf, such as in Figure 27.

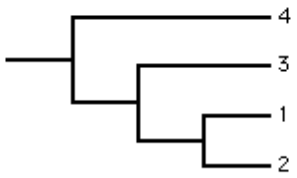


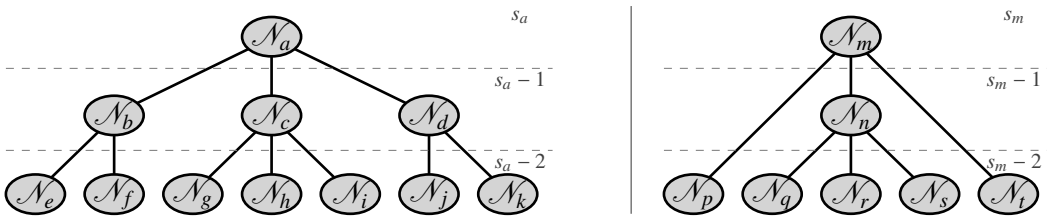
Figure 27: A pectinate tree.

5.2 Cover tree runtime bounds

Of significant interest to the tree community is worst-case runtime bounds for tree-based algorithms. Unfortunately, though, assumption-free worst-case runtime bounds that are better than the brute-force approach continue to elude the community, and this hurdle is not likely to be overcome because of worst-case datasets. However, *adaptive runtime analysis* allows us to show tighter, more descriptive worst-case runtime bounds if we assume that dataset-dependent characteristics do not scale with the data.

To date, the most well-known tree structure to which this technique has been applied is the cover tree [57], which has been mentioned several times already during this thesis and was the subject of the last section. Therefore, refer there for a more in-depth discussion of the cover tree itself, and the expansion constant c, which is the dataset-dependent characteristic that we assume does not scale with the dataset. Importantly, the cover tree has been shown to scale linearly in the dataset size for dual-tree nearest neighbor search and dual-tree kernel density estimation [133]; this is a significant improvement over the quadratic scaling of the brute-force approach.

In this section, we introduce the notion of *cover tree imbalance*, motivated by the previous section's observations on pectinate tree branches. This notion allows us to develop a plug-and-play runtime bound. We will use this plug-and-play bound later, in order to show worst-case linear runtime bounds on dual-tree nearest neighbor search (Section 7.1), dual-tree kernel density estimation (Section 7.3), dual-tree max-kernel search (Section 7.7),



(a) Balanced cover tree.

(b) Imbalanced cover tree.

Figure 28: Balanced and imbalanced cover trees.

dual-tree range search / range count (Section 7.2), and dual-tree kernel matrix approximation (Section 7.5). Each of these bounds is an improvement on the state-of-the-art, and in the case of range search and kernel matrix approximation, are the first such bounds. The work in this section is an adapted and extended version of a recent paper [135].

5.2.1 Tree imbalance

It is well-known that imbalance in trees leads to degradation in performance; for instance, a *kd*-tree node with every descendant in its left child except one is effectively useless. A *kd*-tree full of nodes like this will perform abysmally for nearest neighbor search, and it is not hard to generate a pathological dataset that will cause a *kd*-tree of this sort.

This sort of imbalance applies to all types of trees, not just *kd*-trees. In our situation, we are interested in a better understanding of this imbalance for cover trees, and thus endeavor to introduce a more formal measure of imbalance which is correlated with tree performance. Numerous measures of tree imbalance have already been established; one example is that proposed by Colless [136], and another is Sackin's index [137], but we aim to capture a different measure of imbalance that utilizes the leveled structure of the cover tree.

We already know each node in a cover tree is indexed with an integer level (or scale). In the explicit representation of the cover tree, each non-leaf node has children at a lower level. But these children need not be strictly one level lower; see Figure 28. In Figure 28a, each cover tree node has children that are strictly one level lower; we will refer to this as a

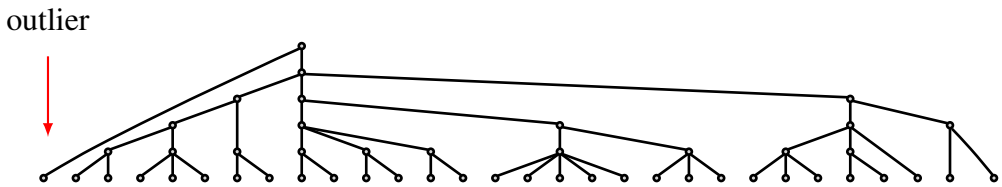


Figure 29: Single-outlier cover tree.

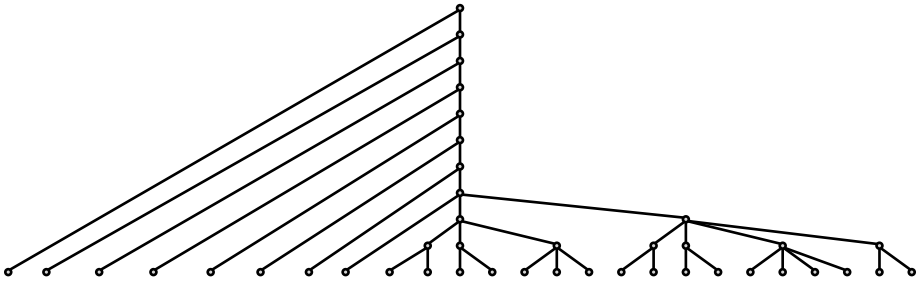


Figure 30: A multiple-outlier cover tree.

perfectly balanced cover tree. Figure 28b, on the other hand, contains the node $\mathcal{N}_m$ which has two children with scale two less than s_m. We will refer to this as an *imbalanced cover tree*. Note that in our definition, the balance of a cover tree has nothing to do with differing number of descendants in each child branch but instead only missing levels.

An imbalanced cover tree can happen in practice, and in the worst cases, the imbalance may be far worse than the simple graphs of Figure 28. Consider a dataset with a single outlier which is very far away from all of the other points[1]. Figure 29 shows what happens in this situation: the root node has two children; one of these children has only the outlier as a descendant, and the other child has the rest of the points in the dataset as a descendant. In fact, it is easy to find datasets with a handful of outliers that give rise to a chain-like structure at the top of the tree: see Figure 30 for an illustration[2].

A tree that has this chain-like structure all the way down, which is similar to the *kd*-tree example at the beginning of this section, is going to perform horrendously; motivated by

[1]Note also that for an outlier sufficiently far away, the expansion constant is $N - 1$, so we should expect poor performance with the cover tree anyway.

[2]As a side note, this behavior is not limited to cover trees, and can happen to mean-split *kd*-trees too, especially in higher dimensions.

this observation, we define a measure of tree imbalance.

Definition 9. *The cover node imbalance $i_n(\mathcal{N}_i)$ for a cover tree node $\mathcal{N}_i$ with scale s_i in the cover tree $\mathcal{T}$ is defined as the cumulative number of missing levels between the node and its parent $\mathcal{N}_p$ (which has scale s_p). If the node is a leaf child (that is, $s_i = -\infty$), then number of missing levels is defined as the difference between s_p and $s_{\min} - 1$ where $s_{\min}$ is the smallest scale of a non-leaf node in $\mathcal{T}$. If $\mathcal{N}_i$ is the root of the tree, then the cover node imbalance is 0. Explicitly written, this calculation is*

$$i_n(\mathcal{N}_i) = \begin{cases} s_p - s_i - 1 & \text{if } \mathcal{N}_i \text{ is not a leaf and not the root node} \\ \max(s_p - s_{\min} - 1,\ 0) & \text{if } \mathcal{N}_i \text{ is a leaf} \\ 0 & \text{if } \mathcal{N}_i \text{ is the root node.} \end{cases} \tag{12}$$

This simple definition of cover node imbalance is easy to calculate, and using it, we can generalize to a measure of imbalance for the full tree.

Definition 10. *The cover tree imbalance $i_t(\mathcal{T})$ for a cover tree $\mathcal{T}$ is defined as the cumulative number of missing levels in the tree. This can be expressed as a function of cover node imbalances easily:*

$$i_t(\mathcal{T}) = \sum_{\mathcal{N}_i \in \mathcal{T}} i_n(\mathcal{N}_i). \tag{13}$$

A perfectly balanced cover tree $\mathcal{T}_b$ with no missing levels has imbalance $i_t(\mathcal{T}_b) = 0$ (for instance, Figure 28a). A worst-case cover tree $\mathcal{T}_w$ which is entirely a chain-like structure with maximum scale $s_{\max}$ and minimum scale $s_{\min}$ will have imbalance $i_t(\mathcal{T}_w) \sim N(s_{\max} - s_{\min})$. Because of this chain-like structure, each level has only one node and thus there are at least N levels; or, $s_{\max} - s_{\min} \geq N$, meaning that in the worst case the imbalance is quadratic in N[3].

[3]Note that in this situation, $c \sim N$ also.

Table 12: Empirically calculated tree imbalances.

		Imbalance		
Dataset	d	$N = 5k$	$N = 50k$	$N = 500k$
lcdm	3	29402	313190	3400305
sdss	4	17691	191716	1483711
power	7	22152	202050	1345825
susy	18	3686	64231	930969
higgs	29	404	1272	73793
covertype	55	9680	100344	1319447
mnist	784	4004	79375	983554

However, for most real-world datasets with the cover tree implementation in **mlpack** [87] and the reference implementation [57], the tree imbalance is near-linear with the number of points. Generally, most of the cover tree imbalance is contributed by leaf nodes whose parent has scale greater than $s_{\min}$. At this time, no cover tree construction algorithm specifically aims to minimize imbalance.

To demonstrate the near-linearity of the cover tree imbalance, Table 12 shows empirically calculated cover tree imbalance for cover trees on a number of datasets built with both the cover tree implementation in **mlpack** [87] and the original reference implementation [57]. The 'power', 'susy', 'higgs', and 'covertype' datasets are found in the UCI Machine Learning Repository [134], the 'mnist' dataset is ubiquitous but absent from the UCI repository for some reason and is published by LeCun et al. [138], and the 'sdss' dataset is Sloan Digital Sky Survey data [139]. Imbalance is shown for subsets of the datasets of size 5000, 50000, and 500000; the actual value of the imbalance is of less important than the scaling properties with N—which tend to be near-linear.

5.2.2 General runtime bound

With the notion of cover tree imbalance developed, we may turn our attention towards a general runtime bound for dual-tree algorithms that use the cover tree. This is the main theoretical result of the entire thesis, and nearly every other theoretical result depends upon

Algorithm 8 The standard pruning dual-tree traversal for cover trees.

Input: query node $\mathscr{N}_q$, set of reference nodes R
Output: none
$s_r^{\max} \leftarrow \max_{\mathscr{N}_r \in R} s_r$
if $(s_q < s_r^{\max})$ **then**
  {Perform a reference recursion.}
  for all $\mathscr{N}_r \in R$ **do**
    `BaseCase(`p_q`,` p_r`)`
  $R_r \leftarrow \{\mathscr{N}_r \in R : s_r = s_r^{\max}\}$
  $R_{r-1} \leftarrow \{\mathscr{C}(\mathscr{N}_r) : \mathscr{N}_r \in R_r\} \cup (R \setminus R_r)$
  $R'_{r-1} \leftarrow \{\mathscr{N}_r \in R_{r-1} : \texttt{Score}(\mathscr{N}_q, \mathscr{N}_r) \neq \infty\}$
  recurse with $\mathscr{N}_q$ and R'_{r-1}
else
  {Perform a query recursion.}
  for all $\mathscr{N}_{qc} \in \mathscr{C}(\mathscr{N}_q)$ **do**
    $R' \leftarrow \{\mathscr{N}_r \in R : \texttt{Score}(\mathscr{N}_q, \mathscr{N}_r) \neq \infty\}$
    recurse with $\mathscr{N}_{qc}$ and R'

it. Although cover trees were originally intended for nearest neighbor search (see Algorithm `Find-All-Nearest` in [57]), they have been adapted to a wide variety of problems: minimum spanning tree calculation [68], approximate nearest neighbor search [67], Gaussian processes posterior calculation [140], and max-kernel search [79] are some examples. We can express the original nearest neighbor search algorithm (and later extensions) inside of the framework of tree-independent dual-tree algorithms, and this gives us the traversal shown in Algorithm 8, originally presented in the context of max-kernel search [79]. This traversal stems from `Find-All-Nearest` by Beygelzimer et al. [57], and is implemented in both the cover tree reference implementation and in a more flexible manner in **mlpack** [87].

Initially, the traversal is called with the root of the query tree and a reference set R containing only the root of the reference tree[4].

This dual-tree recursion is a depth-first recursion in the query tree and a breadth-first

[4]Though this is different than other traversals which take the root of two trees, this still fits in the `TraversalType` abstraction laid out in Section 4.5.3 easily. Also, the implementation of this traversal is *far* more complex than Algorithm 8 might suggest. **mlpack**'s code is highly commented, so the enthusiastic reader is directed to go there to lose their enthusiasm.

recursion in the reference tree; to this end, the recursion maintains one query node $\mathscr{N}_q$ and a reference set R. The set R may contain reference nodes with many different scales; the maximum scale in the reference set is $s_r^{\max}$ (line 3). Each single recursion will descend either the query tree or the reference tree, not both; the conditional in line 4, which determines whether the query or reference tree will be recursed, is aimed at keeping the relative scales of query nodes and reference nodes close.

A query recursion (lines 12–16) is straightforward: for each child $\mathscr{N}_{qc}$ of $\mathscr{N}_q$, the node combinations $(\mathscr{N}_{qc}, \mathscr{N}_r)$ are scored for each $\mathscr{N}_r$ in the reference set R. If possible, these combinations are pruned to form the set R' (line 16) by checking the output of the `Score()` function, and then the algorithm recurses with $\mathscr{N}_{qc}$ and R'.

A reference recursion (lines 4–11) is similar to a query recursion, but the pruning strategy is significantly more complicated. Given R, we calculate R_r, which is the set of nodes in R that have scale $s_r^{\max}$. Then, for each node $\mathscr{N}_r$ in the set of children of nodes in R_r, the node combinations $(\mathscr{N}_q, \mathscr{N}_r)$ are scored and pruned if possible. Those reference nodes that were not pruned form the set R'_{r-1}. Then, this set is combined with $R \setminus R_r$—that is, each of the nodes in R that was *not* recursed into—to produce R', and the algorithm recurses with $\mathscr{N}_q$ and the reference set R'.

The reference recursion only recurses into the top-level subset of the reference nodes in order to preserve the separation invariant. It is easy to show that every pair of points held in nodes in R is separated by at least $2^{s_r^{\max}}$:

Lemma 2. *For all nodes $\mathscr{N}_i, \mathscr{N}_j \in R$ (in the context of Algorithm 8) which contain points p_i and p_j, respectively, $d(p_i, p_j) > 2^{s_r^{\max}}$, with $s_r^{\max}$ defined as in line 3.*

Proof. This proof is by induction. If $|R| = 1$, such as during the first reference recursion, the result obviously holds. Now consider any reference set R and assume the statement of the lemma holds for this set R, and define $s_r^{\max}$ as the maximum scale of any node in R. Construct the set R_{r-1} as in line 9 of Algorithm 8; if $|R_{r-1}| \leq 1$, then R_{r-1} satisfies the desired property.

Otherwise, take any $\mathscr{N}_i, \mathscr{N}_j$ in R_{r-1}, with points p_i and p_j, respectively, and scales s_i and s_j, respectively. Clearly, if $s_i = s_j = s_r^{\max} - 1$, then by the separation invariant $d(p_i, p_j) > 2^{s_r^{\max}-1}$.

Now suppose that $s_i < s_r^{\max} - 1$. This implies that there exists some implicit cover tree node with point p_i and scale $s_r^{\max} - 1$ (as well an implicit child of this node p_i with scale $s_r^{\max} - 2$ and so forth until one of these implicit nodes has child p_i with scale s_i). Because the separation invariant applies to both implicit and explicit representations of the tree, we conclude that $d(p_i, p_r) > 2^{s_r^{\max}} - 1$. The same argument may be made for the case where $s_j < s_r^{\max} - 1$, with the same conclusion.

We may therefore conclude that each point of each node in R_{r-1} is separated by $2^{s_r^{\max}-1}$. Note that $R'_{r-1} \subseteq R_{r-1}$ and that $R \setminus R_{r-1} \subseteq R$ in order to see that this condition holds for all nodes in $R' = R'_{r-1} \cup (R \setminus R_{r-1})$.

Because we have shown that the condition holds for the initial reference set and for any reference set produced by a reference recursion, we have shown that the statement of the lemma is true. □

This observation means that the set of points P held by all nodes in R is always a subset of $C_{s_r^{\max}}$. This fact will be useful in our later runtime proofs.

Next, we develop notions with which to understand the behavior of the cover tree dual-tree traversal when the datasets are of significantly different scale distributions.

If the datasets are similar in scale distribution (that is, inter-point distances tend to follow the same distribution), then the recursion will alternate between query recursions and reference recursions. But if the query set contains points which are, in general, much farther apart than the reference set, then the recursion will start with many query recursions before reaching a reference recursion. The converse case also holds. We are interested in formalizing this notion of scale distribution; therefore, define the following dataset-dependent constants for the query set S_q and the reference set S_r:

- η_q: the largest pairwise distance in S_q

- δ_q: the smallest nonzero pairwise distance in S_q
- η_r: the largest pairwise distance in S_r
- δ_r: the smallest nonzero pairwise distance in S_r

These constants are directly related to the aspect ratio of the datasets; indeed, η_q/δ_q is exactly the aspect ratio of S_q. Further, let us define and bound the top and bottom levels of each tree:

- The *top scale* s_q^T of the query tree $\mathscr{T}_q$ is the scale of the root of $\mathscr{T}_q$, and is bounded as $\lceil \log_2(\eta_q) \rceil - 1 \leq s_q^T \leq \lceil \log_2(\eta_q) \rceil$.
- The *minimum scale* of the query tree $\mathscr{T}_q$ is the scale of the lowest non-leaf node of the tree, and may be explicitly defined equivalently as $s_q^{\min} = \lceil \log_2(\delta_q) \rceil$.
- The top scale s_r^T of the reference tree $\mathscr{T}_r$ is the scale of the root of $\mathscr{T}_r$, and is bounded as $\lceil \log_2(\eta_r) \rceil - 1 \leq s_r^T \leq \lceil \log_2(\eta_r) \rceil$.
- The minimum scale of the reference tree $\mathscr{T}_r$ is the scale of the lowest non-leaf node of the tree, and may be explicitly defined equivalently as $s_r^{\min} = \lceil \log_2(\delta_r) \rceil$.

Note that the minimum scale is not the minimum scale of *any* cover tree node (that would be $-\infty$), but the minimum scale of any non-leaf node in the tree.

Suppose that our datasets are of a similar scale distribution: $s_q^T = s_r^T$, and $s_q^{\min} = s_r^{\min}$. In this setting we will have alternating query and reference recursions. But if this is not the case, then we have extra reference recursions before the first query recursion or after the last query recursion (situations where both these cases happen are possible). Motivated by this observation, let us quantify these extra reference recursions:

Lemma 3. *For a dual-tree algorithm with $S_q \sim S_r \sim O(N)$ using cover trees and the traversal given in Algorithm 8, the number of extra reference recursions that happen before the first query recursion is bounded by*

$$\min\left(O(N), \log_2(\eta_r/\eta_q) - 1\right). \tag{14}$$

Proof. The first query recursion happens once $s_q \geq s_r^{\max}$. The number of reference recursions before the first query recursion is then bounded as the number of levels in the reference tree between s_r^T and s_q^T that have at least one explicit node. Because there are $O(N)$ nodes in the reference tree, the number of levels cannot be greater than $O(N)$ and thus the result holds.

The second bound holds by applying the definitions of s_r^T and s_q^T to the expression $s_r^T - s_q^T - 1$:

$$\begin{aligned} s_r^T - s_q^T - 1 &\leq \lceil \log_2(\eta_r) \rceil - (\lceil \log_2(\eta_q) \rceil - 1) - 1 && (15) \\ &\leq \log_2(\eta_r) + 1 - \log_2(\eta_q) && (16) \end{aligned}$$

which gives the statement of the lemma after applying logarithmic identities. □

Note that the $O(N)$ bound may be somewhat loose, but it suffices for our later purposes. Now let us consider the other case:

Lemma 4. *For a dual-tree algorithm with $S_q \sim S_r \sim O(N)$ using cover trees and the traversal given in Algorithm 8, the number of extra reference recursions that happen after the last query recursion is bounded by*

$$\theta = \max\left\{\min\left[O(N \log_2(\delta_q/\delta_r)), O(N^2)\right],\ 0\right\}. \tag{17}$$

Proof. Our goal here is to count the number of reference recursions after the final query recursion at level $s_q^{\min}$; the first of these reference recursions is at scale $s_r^{\max} = s_q^{\min}$. Because query nodes are not pruned in this traversal, each reference recursion we are counting will be duplicated over the whole set of $O(N)$ query nodes. The first part of the bound follows by observing that $s_q^{\min} - s_r^{\min} \leq \lceil \log_2(\delta_q) \rceil - \lceil \log_2(\delta_r) \rceil - 1 \leq \log_2(\delta_q/\delta_r)$.

The second part follows by simply observing that there are $O(N)$ reference nodes. □

These two previous lemmas allow us a better understanding of what happens as the reference set and query set become different. Lemma 3 shows that the number of extra recursions caused by a reference set with larger pairwise distances than the query set (η_r larger than η_q) is modest; on the other hand, Lemma 4 shows that for each extra level in the reference tree below $s_q^{\min}$, $O(N)$ extra recursions are required. Using these lemmas and this intuition, we will prove general runtime bounds for the cover tree traversal.

Theorem 1. *Given a reference set S_r of size N with an expansion constant c_r and a set of queries S_q of size $O(N)$, a standard cover tree based dual-tree algorithm (Algorithm 8) takes*

$$O\left(c_r^4 |R^*| \chi \psi (N + i_t(\mathscr{T}_q) + \theta)\right) \tag{18}$$

time, where $|R^*|$ is the maximum size of the reference set R (line 1) during the dual-tree recursion, χ is the maximum possible runtime of `BaseCase()`*, ψ is the maximum possible runtime of* `Score()`*, and θ is defined as in Lemma 4.*

Proof. First, split the algorithm into two parts: reference recursions (lines 4–11) and query recursions (lines 12–16). The runtime of the algorithm is bounded as the runtime of a reference recursion times the total number of reference recursions plus the total runtime of all query recursions.

Consider a reference recursion (lines 4–11). Define R^* to be the largest set R for any scale $s_r^{\max}$ and any query node $\mathscr{N}_q$ during the course of the algorithm; then, it is true that $|R| \leq |R^*|$. The work done in the base case loop from lines 6–7 is thus $O(\chi|R|) \leq O(\chi|R^*|)$. Then, lines 9 and 10 take $O(c_r^4 \psi |R|) \leq O(c_r^4 \psi |R^*|)$ time, because each reference node has up to c_r^4 children. So, one full reference recursion takes $O(c_r^4 \psi \chi |R^*|)$ time.

Now, note that there are $O(N)$ nodes in $\mathscr{T}_q$. Thus, line 16 is visited $O(N)$ times. The

amount of work in line 15, like in the reference recursion, is bounded as $O(c_r^4 \psi |R^*|)$. Therefore, the total runtime of all query recursions is $O(c_r^4 \psi |R^*| N)$.

Lastly, we must bound the total number of reference recursions. Reference recursions happen in three cases: *(1)* $s_r^{\max}$ is greater than the scale of the root of the query tree (no query recursions have happened yet); *(2)* $s_r^{\max}$ is less than or equal to the scale of the root of the query tree, but is greater than the minimum scale of the query tree that is not $-\infty$; *(3)* $s_r^{\max}$ is less than the minimum scale of the query tree that is not $-\infty$.

First, consider case *(1)*. Lemma 3 shows that the number of reference recursions of this type is bounded by $O(N)$. Although there is also a bound that depends on the sizes of the datasets, we only aim to show a linear runtime bound, so the $O(N)$ bound is sufficient here.

Next, consider case *(2)*. In this situation, each query recursion implies at least one reference recursion before another query recursion. For some query node $\mathscr{N}_q$, the exact number of reference recursions before the children of $\mathscr{N}_q$ are recursed into is bounded above by $i_n(\mathscr{N}_q) + 1$: if $\mathscr{N}_q$ has imbalance 0, then it is exactly one level below its parent, and thus there is only one reference recursion. On the other hand, if $\mathscr{N}_q$ is many levels below its parent, then it is possible that a reference recursion may occur for each level in between; this is a maximum of $i_n(\mathscr{N}_q) + 1$.

Because each query node in $\mathscr{T}_q$ is recursed into once, the total number of reference recursions before each query recursion is

$$\sum_{\mathscr{N}_q \in \mathscr{T}_q} i_n(\mathscr{N}_q) + 1 = i_t(\mathscr{T}_q) + O(N) \tag{19}$$

since there are $O(N)$ nodes in the query tree.

Lastly, for case *(3)*, we may refer to Lemma 4, giving a bound of θ reference recursions in this case.

We may now combine these results for the runtime of a query recursions with the total number of reference recursions in order to give the result of the theorem:

$$O\left(c_r^4|R^*|\psi\chi\left(N + i_t(\mathscr{T}_q) + \theta\right)\right) + O\left(c_r^4|R^*|\psi N\right) \sim O\left(c_r^4|R^*|\psi\chi\left(N + i_t(\mathscr{T}_q) + \theta\right)\right). \quad (20)$$

□

When we consider the monochromatic case (where $S_q = S_r$), the results trivially simplify.

Corollary 1. *Given the situation of Theorem 1 but with $S_q = S_r = S$ so that $c_q = c_r = c$ and $\mathscr{T}_q = \mathscr{T}_r = \mathscr{T}$, a dual-tree algorithm using the standard cover tree traversal (Algorithm 8) takes*

$$O\left(c^4|R^*|\chi\psi\left(N + i_t(\mathscr{T})\right)\right) \quad (21)$$

time, where $|R^*|$ is the maximum size of the reference set R (line 1) during the dual-tree recursion, χ is the maximum possible runtime of `BaseCase()`*, and ψ is the maximum possible runtime of* `Score()`*.*

An intuitive understanding of these bounds is best achieved by first considering the monochromatic case (this case arises, for instance, in all-nearest-neighbor search). The linear dependence on N arises from the fact that all query nodes must be visited. The dependence on the reference tree, however, is encapsulated by the term $c^4|R^*|$, with $|R^*|$ being the maximum size of the reference set R; this value must be derived for each specific problem. The bad performance of poorly-behaved datasets with large c (or, in the worst case, $c \sim N$) is then captured in both of those terms. Poorly-behaved datasets may also have a high cover tree imbalance $i_t(\mathscr{T})$; the linear dependence of runtime on imbalance is thus sensible for well-behaved datasets.

The bichromatic case ($S_q \neq S_r$) is a slightly more complex result which deserves a bit more attention. The intuition for all terms except θ remain virtually the same.

The term θ captures the effect of query and reference datasets with different widths, and has one unfortunate corner case: when $\delta_q > \eta_r$, then the query tree must be entirely descended before any reference recursion. This results in a bound of the form $O(N \log(\eta_r / \delta_r))$, or $O(N^2)$ (see Lemma 4). This is because the reference tree must be descended individually for each query point.

The quantity $|R^*|$ bounds the amount of work that needs to be done for each recursion. In the worst case, $|R^*|$ can be N. However, dual-tree algorithms rely on branch-and-bound techniques to prune away work (lines 10 and 15 in Algorithm 8). A small value of $|R^*|$ will imply that the algorithm is extremely successful in pruning away work. An (upper) bound on $|R^*|$ (and the algorithm's success in pruning work) will depend on the problem and the data. As we will show, bounding $|R^*|$ is often possible. For many dual-tree algorithms, $\chi \sim \psi \sim O(1)$; often, cached sufficient statistics [125] can enable $O(1)$ runtime implementations of `BaseCase()` and `Score()`.

These results hold for any dual-tree algorithm regardless of the problem. Hence, the runtime of any dual-tree algorithm would be at least $O(N)$ using our bound, which matches the intuition that answering $O(N)$ queries would take at least $O(N)$ time. For a particular problem and data, if c_r, $|R^*|$, χ, ψ are bounded by constants independent of N and θ is no more than linear in N (for large enough N), then the dual-tree algorithm for that problem has a runtime linear in N. Our theoretical result separates out the problem-dependent and the problem-independent elements of the runtime bound, which allows us to simply plug in the problem-dependent bounds in order to get runtime bounds for any dual-tree algorithm without requiring an analysis from scratch.

Our results are similar to that of Ram et al. [133], but those results depend on a quantity called the *constant of bichromaticity*, denoted κ, which has unclear relation to cover tree imbalance. The dependence on κ is given as $c_q^{4\kappa}$, which is not a good bound, especially because κ may be much greater than 1 in the bichromatic case (where $S_q \neq S_r$).

The more recent results for max-kernel search [79] are more related to these results,

but they depend on the *inverse constant of bichromaticity* ν which suffers from the same problem as κ. Although the dependence on ν is linear (that is, $O(\nu N)$), bounding ν is difficult and it is not true that $\nu = 1$ in the monochromatic case.

ν corresponds to the maximum number of reference recursions between a single query recursion, and κ corresponds to the maximum number of query recursions between a single reference recursion. The respective proofs that use these constants then apply them as a worst-case measure for the whole algorithm: when using κ, Ram et al. [133] assume that *every* reference recursion may be followed by κ query recursions; similarly, myself and Ram [79] assume that *every* query recursion may be followed by ν reference recursions. In this proof, though, we have simply used $i_t(\mathscr{T}_q)$ and θ as an exact summation of the total extra reference recursions, which gives us a much tighter bound than ν or κ on the running time of the whole algorithm.

Further, both ν and κ are difficult to empirically calculate and require an entire run of the dual-tree algorithm. On the other hand, bounding $i_t(\mathscr{T}_q)$ (and θ) can be done in one pass of the tree (assuming the tree is already built). Thus, not only is our bound tighter when the cover tree imbalance is sublinear in N, it more closely reflects the actual behavior of dual-tree algorithms, and the constants which it depends upon are straightforward to calculate.

Later in the thesis, we will apply Theorem 1 to many different algorithms and see how it simplifies proofs and provides an intuitive and useful bound.

5.3 An issue with the cover tree single-tree runtime bound proof

Beygelzimer, Kakade, and Langford show a proof that claims an $O(c^{12} \log N)$ worst-case runtime per query for single-tree nearest neighbor search using cover trees [57]. This sub-linear runtime bound has been adapted to other situations [46, 67] and referenced repeatedly [127, 141, 142, 143, 144] despite the fact that there is a flaw in the proof. Let us review

Algorithm 9 Single-tree nearest neighbor search for cover trees.

1: **Input**: query point p_q, reference tree $\mathscr{T}_i$
2: **Output**: nearest neighbor $\hat{p}_{nn}$
3: $R \leftarrow \{\text{root}(\mathscr{T}_i)\}$
4: **while** $|R| > 0$ **do**
5: {Form new reference set out of children of nodes with maximum scale.}
6: $s_r^{\max} \leftarrow$ maximum scale of nodes in R
7: $R' \leftarrow \{\mathscr{N}_r \in R : s_r = s_r^{\max}\}$
8: $R'' \leftarrow \bigcup_{\mathscr{N}_r \in R'} \mathscr{C}_r$
9: {Determine the nearest neighbor in R'' and attempt to prune.}
10: {(A tree-independent formulation would have `Score()` and `BaseCase()` here.)}
11: $\hat{p}_{nn} \leftarrow \text{argmin}_{\mathscr{N}_i \in R''} d(p_q, p_i)$
12: $R_{s_r^{\max}-1} \leftarrow \{\mathscr{N}_i \in R'' : d(p_q, p_i) \leq d(p_q, \hat{p}_{nn}) + 2^{s_r^{\max}}$
13: {Merge unpruned nodes into R and remove nodes we recursed into.}
14: $R \leftarrow (R \setminus R') \cup R_{s_r^{\max}-1}$
15: **return** $\hat{p}_{nn}$

the theorem and proof [57] in order to discuss the flaw. Algorithm 9 introduces a non-tree-independent formulation of the single-tree nearest neighbor search algorithm for cover trees. It is straightforward to generalize, but for the purposes of this short discussion there is no need.

When Algorithm 9 is called with a query point p_q and reference tree $\mathscr{T}_r$, a breadth-first traversal of $\mathscr{T}_r$ is performed, pruning branches when possible (in line 12).

Theorem 2 (Theorem 5 from Beygelzimer, Kakade, and Langford [57])**.** *If the dataset $S_r \cup \{p_q\}$ has expansion constant c and $|S_r| = N$, the nearest neighbor of p_q can be found in time $O(c^{12} \log N)$.*

In order to discuss the issue with the proof, we need to present the proof until the flaw (no more is necessary).

Partial proof. (Notation adapted from Beygelzimer, Kakade, and Langford [57]) Let R^* be the last R considered by the algorithm (so, R^* consists only of leaf nodes with scale $-\infty$). Lemma 4.3 (from [57]) bounds the explicit depth of any node in the tree (and in particular

any node in R^*) by $k = O(c^2 \log N)$. Consequently, the number of iterations is at most $k|R^*| \leq k$. □

This assertion attempts to bound the number of iterations in the **while** loop by multiplying the size of the final reference set R^* by the depth bound $O(c^2 \log N)$. This does effectively bound all of the iterations required by the ancestors of every node in R^*, but it potentially undercounts. Suppose that one branch of the tree $\mathscr{T}_r$ has nodes with scales that are not present elsewhere in the tree (call this the *special branch*, just for clarity). Suppose now that all descendant nodes of this special branch are pruned before the final iteration (so, R^* contains no nodes from this particular branch). The bounding strategy using $|R^*|O(c^2 \log N)$ iterations does not count any iterations caused by the nodes in the special branch!

This flaw in the proof means that the result may potentially be incorrect, and there is no easy way to resolve the issue of uncounted iterations and retain the sublinear runtime bound. Although it is surely true that scaling logarithmic in N will be observed in practice for well-behaved datasets, the assumption that c does not change as the dataset grows does not necessarily mean that sublinear search times are guaranteed.

CHAPTER 6

TRAVERSALS

This chapter is focused on improvement of the second piece of dual-tree algorithms: the traversals. As introduced in Chapter 3, the dual-tree traversal visits combinations of nodes in the query and reference tree, and calls a `Score()` function to determine if this node combination can be pruned. If not, a `BaseCase()` function is called on each pair of query and reference points held in the nodes, and the node combination is recursed into. It is worth re-printing the definition here:

Definition 11. *A pruning dual-tree traversal is a process that, given two space trees $\mathscr{T}_q$ (the query tree, built on the query set S_q) and $\mathscr{T}_r$ (the reference tree, built on the reference set S_r), will visit combinations of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ such that $\mathscr{N}_q \in \mathscr{T}_q$ and $\mathscr{N}_r \in \mathscr{T}_r$ no more than once, and call a function `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` to assign a score to that node. If the score is ∞, the combination is pruned and no combinations $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ such that $\mathscr{N}_{qc} \in \mathscr{D}_q^n$ and $\mathscr{N}_{rc} \in \mathscr{D}_r^n$ are visited. Otherwise, for every combination of points (p_q, p_r) such that $p_q \in \mathscr{P}_q$ and $p_r \in \mathscr{P}_r$, a function `BaseCase(`p_q`,` p_r`)` is called. If no node combinations are pruned during the traversal, `BaseCase(`p_q`,` p_r`)` is called at least once on each combination of $p_q \in S_q$ and $p_r \in S_r$.*

Common strategies for traversals are dual breadth-first traversals and dual depth-first traversals. More complex strategies are possible: the standard cover tree traversal, introduced in Algorithm 8, is a combination of breadth-first and depth-first.

This chapter discusses an improved dual depth-first traversal that is shown to provide good speedup for the task of nearest neighbor search. This traversal is also applicable to other tasks, though, because it is presented in a tree-independent manner. This presentation of the improved dual depth-first traversal is based on recently submitted work [66].

6.1 Improved dual depth-first traversal

Algorithm 10 `DualDepthFirstTraversal(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`).`

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: none

{Perform base cases for points in node combination.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    `BaseCase(`p_q`,` p_r`)`

{Assemble list of combinations to recurse into.}
$q \leftarrow$ empty priority queue
if $\mathscr{N}_q$ and $\mathscr{N}_r$ both have children **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
      $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_{rc}$`)`
      if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ has children but $\mathscr{N}_r$ does not **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_r$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ does not have children but $\mathscr{N}_r$ does **then**
  for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_q$,$\mathscr{N}_{rc}$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_q, \mathscr{N}_{rc})$ into q with priority $1/s_i$

{Recurse into combinations with highest priority first.}
for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}) \in q$, highest priority first **do**
  `DualDepthFirstTraversal(`$\mathscr{N}_{qi}$`,` $\mathscr{N}_{ri}$`)`

Although the definition is quite complex, real-world dual-tree traversals tend to be straightforward. The standard depth-first dual-tree traversal is shown in Algorithm 10; this is the same traversal used in most dual-tree algorithms that use the *kd*-tree [61] [68] [74][1] and is often used in practice [87]. Generally, a depth-first traversal is preferred because many space trees in practice only hold points in the leaves; breadth-first traversals may not perform well in these situations. To illustrate this, consider the example of nearest neighbor search where the pruning depends on the current candidate nearest neighbors; a breadth-first search will not encounter any points until the leaves of the tree, and therefore nothing

[1]The algorithms in each of the referenced papers tend to look very different because they are not derived in a tree-independent form, but using the *kd*-tree with the traversal in Algorithm 10 and simplifying will yield the same algorithm.

can be pruned because no candidate nearest neighbors will have been encountered[2].

The traversal is originally called with the root of the query tree $\mathscr{T}_q$ and the root of the reference tree $\mathscr{T}_r$. First, `BaseCase()` is called with every pair of query and reference points (lines 4–6)—note that this is *not* every pair of *descendant* query and reference points. Then, for recursion, we collect a list of combinations to recurse into, sorted by their score. Any combinations with score ∞ are not recursed into. If both nodes have children, then we recurse into combinations of query children and reference children. If only the reference node has children, we recurse into combinations of the query node and the reference children. If only the query node has children, we recurse into combinations of the query children and the reference node. If neither node has children, there is no need to recurse.

The algorithm first recurses into those node combinations with lowest score. Depending on the task being solved (that is, which `Score()` and `BaseCase()` functions are being used), this prioritized approach to recursion can provide significant speedup over unprioritized recursion. For nearest neighbor search, a prioritized recursion gives significantly faster results. Other dual-tree algorithms that would also see speedup include max-kernel search, minimum spanning tree calculation, kernel density estimation, and k-means clustering.

6.1.1 Prioritized recursions and nearest neighbor search

Algorithm 10 is the standard depth-first dual-tree traversal that is used in practice, and it prioritizes recursions: node combinations with lower scores (from `Score()`) are recursed into first. Therefore, let us consider a task that is benefitted by a prioritized recursion: nearest neighbor search (described earlier in Section 2.2). The goal is, for each query point p_q in the query set S_q, find the nearest point in the reference set S_r.

Algorithms 11 and 12 present simplified versions of the `BaseCase()` and `Score()` later presented in Section 7.1. This algorithm is just a tree-independent expression of the

[2]It should be noted that it is possible to generate pruning rules for nearest neighbor search that can work for a breadth-first traversal also, and this is done later in Section 7.1. But rules that complex are not always preferred in practice.

Algorithm 11 Simple `BaseCase()` for nearest neighbor search.

Input: query point p_q, reference point p_r, candidate point $N[p_q]$, candidate distance $D[p_q]$
Output: distance $d(p_q, p_r)$
if $d(p_q, p_r) < D[p_q]$ **then**
    $N[p_q] \leftarrow p_r$
    $D[p_q] \leftarrow d(p_q, p_r)$

Algorithm 12 Simple `Score()` for nearest neighbor search.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ or ∞ if it should be pruned
if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > B_{df}(\mathscr{N}_q)$ **then**
    return ∞
return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$

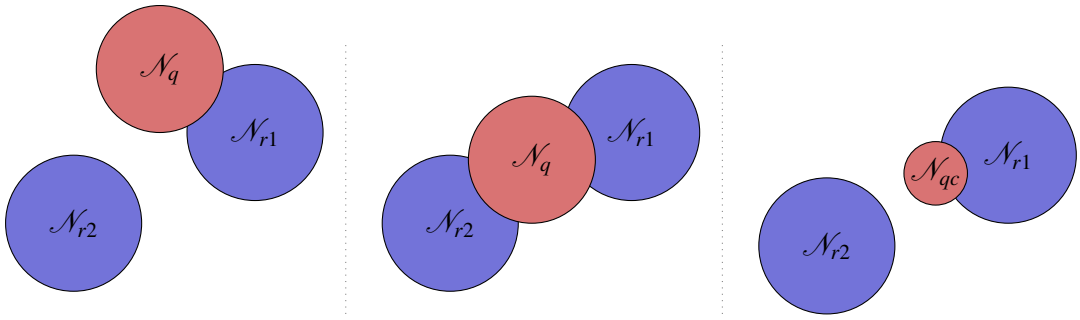
nearest neighbor search algorithm given in Algorithm 3 from Section 2.6.

The algorithm maintains a list of candidate neighbors ($N[\cdot]$) and a list of candidate distances ($D[\cdot]$). During the algorithm, for a given query point p_q, $N[p_q]$ contains the current nearest neighbor candidate of p_q, and $D[p_q]$ contains the distance between p_q and its current nearest neighbor candidate. At the end of the algorithm, $N[p_q]$ contains the nearest neighbor of p_q. For initialization, $D[p_q]$ is set to ∞ (or some other sufficiently large value).

The bound function, $B_{df}(\mathscr{N}_q)$, represents the worst candidate nearest neighbor distance for any descendant point of the query node: $\max_{p_q \in \mathscr{D}_q^p} D[p_q]$. However, this formulation would require looping over every descendant point in $\mathscr{N}_q$, so it is infeasible to calculate. Fortunately, we can use caching along with a re-expression of the worst candidate nearest neighbor distance to define $B_{df}(\mathscr{N}_q)$ in a way we can easily calculate:

$$B_{df}(\mathscr{N}_q) = \max \left\{ \max_{p_q \in \mathscr{P}_q} D[p_q], \max_{\mathscr{N}_{qc} \in \mathscr{C}_q} B_{df}(\mathscr{N}_{qc}) \right\}. \tag{22}$$

We may cache the current value of $B_{df}(\cdot)$ in any query node, and then subsequent calculations of $B_{df}(\cdot)$ include only a scan over the points held in the node and over the children held by the node. In general this is a very fast, $O(1)$ calculation: *kd*-trees, for instance, have



(a) $\mathcal{N}_q$ closer to $\mathcal{N}_{r1}$. (b) $\mathcal{N}_q$ equidistant... (c) ...but $\mathcal{N}_{qc}$ is not equidistant.

Figure 31: Different situations for recursion.

no more than two children, and only hold a specified number of points in the leaves.

Now, see that the result of `Score()` is $d_{\min}(\mathcal{N}_q, \mathcal{N}_r)$, if the node is not pruned. This encourages the traversal to first recurse into node combinations where the query and reference node are closer; the intuition here is that for a given query node, recursing into a closer reference node is more likely to produce closer nearest neighbor candidates for each query descendant point, thus allowing $B_{df}(\cdot)$ to tighten more and more work to be pruned as a result. Especially for a depth-first recursion near the top of the tree, the stakes are high: a bad recursion choice can potentially mean huge amounts of extra work.

6.1.2 Delaying reference recursion

In the situation depicted in Figure 31a, combination $(\mathcal{N}_q, \mathcal{N}_{r1})$ should be visited before combination $(\mathcal{N}_q, \mathcal{N}_{r2})$. It is clear that this is the right choice, because a depth-first traversal of $(\mathcal{N}_q, \mathcal{N}_{r1})$ is more likely to tighten the bound $B_{df}(\mathcal{N}_q)$ such that $(\mathcal{N}_q, \mathcal{N}_{r2})$ can be pruned when it is recursed into.

But, consider a more tricky case, depicted in Figure 31b. Here, $d_{\min}(\mathcal{N}_q, \mathcal{N}_{r1}) = d_{\min}(\mathcal{N}_q, \mathcal{N}_{r2}) = 0$, so we are unable to tell whether it is better to recurse into $(\mathcal{N}_q, \mathcal{N}_{r1})$ first or into $(\mathcal{N}_q, \mathcal{N}_{r2})$ first. Indeed, Algorithm 10 will select arbitrarily. This situation may occur in Algorithm 10 from lines 11 to 13 if, for a given child query node $\mathcal{N}_{qc}$, two or more reference children $\mathcal{N}_{rc}$ have the same score s_i.

We can do better than arbitrary selection. Consider some child $\mathcal{N}_{qc}$ of $\mathcal{N}_q$. Figure

31c shows an example $\mathscr{N}_{qc}$. In this example, the choice is now clear: the combination $(\mathscr{N}_{qc}, \mathscr{N}_{r1})$ should be recursed into before $(\mathscr{N}_{qc}, \mathscr{N}_{r2})$. Thus, the correct answer to the question "should we recurse into $(\mathscr{N}_q, \mathscr{N}_{r1})$ or $(\mathscr{N}_q, \mathscr{N}_{r2})$ first?" is to sidestep the question entirely: we should not recurse in the reference node, but instead in the query node. Then, at the level of the query child, the decision may be clearer.

A clever reader may ask, "Why not use the distance between the centers of the query node and reference node as the score? This would alleviate many situations like that in Figure 31b." Although this is true, it is sidestepping the issue. Consider Figure 31b: if $\mathscr{N}_q$ has two children, and one is closer to $\mathscr{N}_{r1}$ (like $\mathscr{N}_{qc}$ in Figure 31c) and the other is closer to $\mathscr{N}_{r2}$, then regardless of which reference node is chosen for recursion, the choice will be suboptimal for one of the two query children. Therefore, it is more prudent to wait, and recurse in the query node one more level before making a decision.

In essence, the strategy is to delay recursion in the reference nodes until it is clear which reference node should be recursed into first. This improvement, once generalized, is encapsulated in Algorithm 13. Lines 15–20 check if reference recursion should be delayed because the scores of all reference children are identical. If so, the recursion will proceed by recursing only in the queries. If necessary, this reference recursion delay will continue until no longer possible. This delay is not possible when the query node does not have any children. This improved strategy can make a huge difference in the performance of the algorithm; recursing into a suboptimal reference child first can cause the bound $B_{df}(\cdot)$ to be unnecessarily loose, whereas first recursing into the best reference child will tighten $B_{df}(\cdot)$ more quickly and possibly allow other reference children to be pruned entirely.

For trees such as the *kd*-tree where each node has two children only, the extra implementation overhead for this strategy is trivial and simplifies to the addition of a single `if` statement. However, note that there are some situations where the modified traversal will not outperform the original prioritized traversal. For instance, for nearest neighbor search, if the query tree is identical to the reference tree and nodes in the tree cannot overlap, then

Algorithm 13 `ImprovedDualDepthFirstTraversal(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`).`

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: none

{Perform base cases for points in node combination.}
for all $p_q \in \mathscr{P}_q$ **do**
  for all $p_r \in \mathscr{P}_r$ **do**
    `BaseCase(`p_q`,` p_r`)`

{Assemble list of combinations to recurse into.}
$q \leftarrow$ empty priority queue
if $\mathscr{N}_q$ and $\mathscr{N}_r$ both have children **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $q_{qc} \leftarrow \{\}$
    for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
      $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_{rc}$`)`
      if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_{rc}, s_i)$ into q_{qc}
    if all elements of q_{qc} have identical score **then**
      $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_r$`)`
      push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
    else
      for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}, s_i) \in q_{qc}$ **do**
        push $(\mathscr{N}_{qi}, \mathscr{N}_{ri})$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ has children but $\mathscr{N}_r$ does not **then**
  for all $\mathscr{N}_{qc} \in \mathscr{C}_q$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_{qc}$,$\mathscr{N}_r$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_{qc}, \mathscr{N}_r)$ into q with priority $1/s_i$
else if $\mathscr{N}_q$ does not have children but $\mathscr{N}_r$ does **then**
  for all $\mathscr{N}_{rc} \in \mathscr{C}_r$ **do**
    $s_i \leftarrow$ `Score(`$\mathscr{N}_q$,$\mathscr{N}_{rc}$`)`
    if $s_i \neq \infty$ **then** push $(\mathscr{N}_q, \mathscr{N}_{rc})$ into q with priority $1/s_i$

{Recurse into combinations with highest priority first.}
for all $(\mathscr{N}_{qi}, \mathscr{N}_{ri}) \in q$, highest priority first **do**
  `ImprovedDualDepthFirstTraversal(`$\mathscr{N}_{qi}$`,` $\mathscr{N}_{ri}$`)`

it is very unlikely that the situation described in Figure 31a will be encountered: during the recursion, the query node will only overlap itself and possibly be adjacent to a sibling node.

Table 13: Dataset information.

Dataset	n	d
cloud	2048	10
winequality	6497	11
birch3	100000	2
miniboone	130064	50
covertype	581012	55
power	2075259	7
lcdm	16777216	3
sdss-dr6	39761242	4

6.1.3 Experimental evalution

To test the efficiency of this strategy, we will observe the performance of our recursion strategy on the tasks of exact and approximate nearest neighbor search, with multiple types of trees, and with many different datasets. For approximate search, we compare with LSH (locality-sensitive hashing). The datasets utilized in these experiments are described in Table 13. Each dataset is from the UCI dataset repository [134], with the exception of the birch3 dataset [145], LCDM dataset [146], and SDSS-DR6 dataset [147].

The first test will focus on the task of exact nearest neighbor search; Algorithms 11 and 12 paired with a type of tree and traversal. Using the flexible **mlpack** library [87], we test with the *kd*-tree and the ball tree, using three dual-tree traversal strategies: a depth-first unordered recursion (equivalent to Algorithm 10 where the recursion priority is ignored); the standard depth-first prioritized recursion (Algorithm 10); and our improved recursion (Algorithm 13). In addition, a single-tree algorithm is used; this is the canonical tree-based nearest neighbor search algorithm [31] with a prioritized recursion, run once for each query point. The dataset is randomly split into 60% reference set and 40% query set, and the algorithm is run ten times. The number of distance evaluations and the total runtime are collected. Table 14 shows the average number of distance calculations for each algorithm and the average runtime for each algorithm.

We can see from the results that our improvement is, in many cases, significant. In

Table 14: Runtime (distance evaluations) for exact nearest neighbor search.

algorithm	cloud	winequality	birch3	miniboone
kd-tree, unordered	0.036s (270k)	0.288s (2.15M)	7.310s (62.2M)	62.481s (214M)
kd-tree, prioritized	0.005s (34.2k)	0.039s (222k)	0.419s (2.90M)	25.081s (78.8M)
kd-tree, improved	0.005s (**27.7k**)	0.021s (**104k**)	0.201s (**1.10M**)	12.643s (34.5M)
single *kd*-tree	0.005s (32.9k)	**0.017s** (112k)	0.262s (1.65M)	**6.637s (19.2M)**
ball tree, unordered	0.011s (356k)	0.104s (3.08M)	1.817s (71.6M)	32.947s (616M)
ball tree, prioritized	0.003s (104k)	0.023s (666k)	0.285s (10.9M)	27.934s (514M)
ball tree, improved	0.003s (86.8k)	0.017s (455k)	**0.160s** (5.65M)	**2.332s** (351M)
single ball tree	**0.002s** (69.6k)	**0.012s** (315k)	0.165s (**5.38M**)	26.357s (**254M**)

algorithm	covertype	power	lcdm	sdss-dr6
kd-tree, unordered	302.8s (1.09B)	1163.0s (18.7B)	5628.7s (41.5B)	24717s (156B)
kd-tree, prioritized	15.823s (52.5M)	30.072s (302M)	319.871s (1.87B)	9069s (50.3B)
kd-tree, improved	**4.469s (12.8M)**	**12.714s (200M)**	**71.587s (350M)**	**428.9s (2.14B)**
single *kd*-tree	6.207s (16.3M)	19.684s (232M)	120.6s (476M)	471.4s (2.24B)
ball tree, unordered	163.027s (2.90B)	771.975s (25.3B)	1861.9s (71.1B)	9444s (363B)
ball tree, prioritized	52.487s (902M)	113.437s (3.90B)	386.74s (14.4B)	5202s (192B)
ball tree, improved	**27.251s** (392M)	**83.744s** (2.58B)	**195.175s** (6.46B)	**5150s** (136B)
single ball tree	29.948s (**228M**)	138.422s (**2.49B**)	402.6s (**5.93B**)	7226s (**101B**)

the best case, it gives more than 2x speedup over the next fastest strategy. This effect is especially pronounced on larger datasets, which will have deeper trees: a bad recursion decision early on can significantly affect the ability to prune during the algorithm. Ball trees exhibit less pronounced effects. This is because the bounding structure is a ball of fixed radius, whereas the *kd*-tree is adaptive in all dimensions. Therefore, two child nodes of a ball tree node may overlap, causing the improved strategy of delaying reference recursions to not pay off at lower levels. Nonetheless, especially for large datasets, where the dual-tree strategy is faster than the single-tree strategy, the improved traversal is a clear best choice.

The second task is approximate nearest neighbor search, and in this situation we will also be able to compare with locality-sensitive hashing. Relative-value approximation means that for an approximation parameter ϵ, we are guaranteed for a query point p_q with true nearest neighbor p_r^*, the algorithm will return an approximate nearest neighbor $\hat{p}_r$ such that $d(p_q, \hat{p}_r) \leq (1 + \epsilon) d(p_q, p_r^*)$. It is easy to modify the given `Score()` function to enforce this condition; replace the equation in line 3 of Algorithm 12 with

Table 15: Runtime (distance calculations) [ϵ or M/W] for approximate NN search.

algorithm	cloud	winequality	birch3	miniboone
kd-tree, unordered	0.005s (34.5k) [1.5]	0.025s (148k) [1.44]	0.267s (2.14M) [1.44]	6.831s (22.6M) [1.38]
kd-tree, prioritized	0.003s (17.4k) [1.5]	0.012s (74.5k) [1.5]	0.140s (1.16M) [1.5]	4.863s (15.5M) [1.38]
kd-tree, improved	**0.002s (13.7k)** [1.7]	**0.010s (51.2k)** [1.63]	**0.107s (654k)** [1.63]	3.360s (9.28M) [1.38]
single *kd*-tree	0.003s (23.2k) [2.45]	0.013s (78.0k) [2.33]	0.198s (1.47M) [2.33]	**1.845s (5.75M)** [1.5]
ball tree, unordered	**0.002s** (50.8k) [27.6]	0.007s (186k) [32.3]	0.079s (2.72M) [11.5]	**2.942s** (50.4M) [285]
ball tree, prioritized	**0.002s** (49.2k) [27.6]	**0.006s** (167k) [32.3]	**0.072s** (2.46M) [11.5]	3.266s (54.2M) [249]
ball tree, improved	**0.002s** (45.1k) [27.6]	**0.006s (161k)** [32.3]	**0.072s (2.25M)** [11.5]	3.494s (50.3M) [99]
single ball tree	**0.002s** (43.2k) [999]	**0.006s** (176k) [36.0]	0.111s (3.56M) [10.1]	3.812s (**36.1M**) [99]
multiprobe LSH	0.031s (19.3k) [20/122]	0.011s (472k) [37/33]	1.614s (8.85M) [8/16k]	175.995s (1.77B) [13/328]

algorithm	covertype	power	lcdm	sdss-dr6
kd-tree, unordered	7.796s (27.4M) [1.5]	419.725s (13.0B) [1.27]	75.432s (508M) [1.33]	512.829s (2.89B) [1.27]
kd-tree, prioritized	2.954s (10.6M) [1.5]	**8.392s (189M)** [1.44]	44.187s (306M) [1.38]	380.047s (2.17B) [1.27]
kd-tree, improved	**2.045s (6.25M)** [1.5]	11.044s (191M) [1.56]	**29.069s (160M)** [1.44]	**242.624s (1.11B)** [1.27]
single *kd*-tree	3.869s (11.2M) [1.86]	16.674s (226M) [2.33]	85.821s (397M) [1.78]	329.663s (1.58B) [1.27]
ball tree, unordered	2.187s (33.0M) [99]	415.964s (13.0B) [11.5]	**19.776s** (668M) [19]	**73.638s** (239M) [49]
ball tree, prioritized	**2.183s (32.3M)** [75.9]	**6.753s (233M)** [13.3]	20.158s (**660M**) [19]	75.687s (**237M**) [49]
ball tree, improved	2.539s (33.8M) [49]	8.269s (248M) [15.7]	25.749s (702M) [21.2]	299.8s (451M) [49]
single ball tree	5.496s (40.3M) [27.6]	19.097s (431M) [15.7]	113.299s (1.46B) [21.2]	2054.8s (3.06B) [19]
multiprobe LSH	130.699s (963M) [0.51]	1181.32s (14.0B) [63/9.6]	*timeout* [14/0.968]	*timeout* [7/0.29]

$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > (1/(1+\epsilon))B(\mathscr{N}_q)$.

After applying this change, testing is performed in the same way as for exact nearest neighbor search. ϵ for each tree-based approach is selected to give an average per-point relative error of 0.1 ($\pm$0.01) for each dataset. Because our scheme does not allow the error for an individual point to exceed ϵ, the actual relative error for an individual query point is often much lower. Thus, it is often necessary to set ϵ far higher than the target average error of 0.1. For LSH, the LSHKIT package is used, which implements multi-probe LSH and autotunes the hashing parameters [148]. We use the suggested number of hash tables ($L = 10$) and probes ($T = 20$), and then autotune to select the number of hash functions (M) and bin width (W). Autotuning failed for the larger power, lcdm, and sdss-dr6 datasets; in these cases suggestions of the LSHKIT authors are used [149].

The results are given in Table 15. With approximation, the improved dual-tree traversal performs fewer distance calculations on smaller datasets, and is still dominant for the larger datasets with *kd*-trees. LSH is not competitive on the larger datasets, and on the largest datasets LSH did not complete within 3 days, but it should be noted that the low-dimensional setting is where trees are most effective.

Overall, for large datasets in low-to-medium dimensions, dual-tree search is faster, and the improved traversal we have proposed is the fastest. These experiments seem to show for smaller datasets, single-tree search may be fastest; for sufficiently high dimensions, LSH is faster. This corroborates existing results [125]; as the dimension of data gets higher, pruning rules become less effective. Regardless, in low-to-medium dimensions, the improved dual-tree traversal is dominant.

CHAPTER 7

ALGORITHMS

This chapter, certainly the longest in the thesis, shows how we can use the versatile tree-independent dual-tree algorithm abstraction to describe numerous algorithms—and develop entirely new ones—using only a `BaseCase()` and `Score()` function. Some of the algorithms presented here were not originally designed by me but I have generalized them from their tree-specific form to an actual tree-independent algorithm; others I have improved somewhat; others still are completely original contributions. For some algorithms, I have used the theoretical adaptive analysis techniques presented in Section 5.2 in order to bound the running time of the algorithm.

7.1 Nearest neighbor search

Nearest neighbor search is arguably the most well-known problem to be solved by dual-tree algorithms, with multiple dual-tree algorithms proposed for both exact and approximate nearest-neighbor search [61, 67, 28] as well as a nearly infinite set of other techniques for exact and approximate solutions [150, 31, 33, 50, 151, 152, 153, 57, 56, 154, 155, 120, 156, 157, 158, 32]—and the citations here represent only a miniscule fraction of the not-completely-connected graph of nearest neighbor search literature.

Here, we will allow ourselves to consider the slightly more general problem of k-nearest neighbor search, where instead of finding only one neighbor, we find k neighbors for each query point. To formalize the problem, we can state it as follows:

Given a query dataset S_q, a reference dataset S_r, and an integer $k : 0 < k < N$, for each point $p_q \in S_q$, find the k nearest neighbors in S_r and their distances from p_q. The list of nearest neighbors for a point p_q can be referred to as N_{p_q} and the distances to nearest neighbors for p_q can be referred to as D_{p_q}. Thus, the k-th nearest neighbor to point p_q is $N_{p_q}[k]$ and $D_{p_q}[k] = \|p_q - N_{p_q}[k]\|$.

Algorithm 14 k-nearest-neighbors `BaseCase()`

Input: query point p_q, reference point p_r, list of k nearest candidate points N_{p_q} and k candidate distances D_{p_q} (both ordered by ascending distance)
Output: distance d between p_q and p_r

$d \leftarrow \|p_q - p_r\|$

if $d < D_{p_q}[k]$ **and** `BaseCase(`p_q`,` p_r`)` not yet called **then**
  insert d into ordered list D_{p_q} and truncate list to length k
  insert p_r into N_{p_q} such that N_{p_q} is ordered by distance and truncate list to length k
return d

Clearly, a brute-force approach can be used: compare every possible point combination and store the k smallest distance results for each p_q. Of course, this scales poorly—hence the vast collection of literature referenced above. We will add to this vast collection by describing our own tree-independent dual-tree algorithm to solve the k-nearest neighbor search task. Notation used in this section is given in Chapter 3; specifically, in Table 1.

7.1.1 A tree-independent dual-tree algorithm

We unify all of these branch-and-bound strategies by defining methods `BaseCase(`p_q`,` p_r`)` and `Score(`$\mathcal{N}_q$`,` $\mathcal{N}_r$`)` for use with a pruning dual-tree traversal.

At the initialization of the tree traversal, the lists N_{p_q} and D_{p_q} are empty lists for each query point p_q. After the traversal is complete, the set $\{N_{p_q}[1], ..., N_{p_q}[k]\}$ is the ordered set of k nearest neighbors of the query point p_q, and each $D_{p_q}[i] = \|p_q - N_{p_q}[i]\|$. If we assume that $D_{p_q}[i] = \infty$ if i is greater than the length of D_{p_q}, we can formulate `BaseCase()` as given in Algorithm 14[1].

With the base case established, only the pruning rule remains. A valid pruning rule will, for a given query node $\mathcal{N}_q$ and reference node $\mathcal{N}_r$, prune the reference subtree rooted at $\mathcal{N}_r$ if and only if it is known that there are no points in $\mathcal{D}_r^p$ that are in the set of k nearest neighbors of any points in $\mathcal{D}_q^p$. Thus, at any point in the traversal, we can prune the combination $(\mathcal{N}_q, \mathcal{N}_r)$ if and only if $d_{\min}(\mathcal{N}_q, \mathcal{N}_r) \geq B_1(\mathcal{N}_q)$, where

[1] In practice, k-nearest-neighbors is often run with identical reference and query sets. In that situation it may be useful to modify this implementation of `BaseCase()` so that a point does not return itself as the nearest neighbor (with distance 0).

$$B_1(\mathscr{N}_q) = \max_{p \in \mathscr{D}_q^p} D_p[k]. \tag{23}$$

Now, we can describe this bound recursively. This is important for implementation; a recursive function can cache previous calculations for large speedups.

$$\begin{aligned} B_1(\mathscr{N}_q) &= \max\{\max_{p \in \mathscr{P}_q} D_p[k], \max_{p \in \mathscr{D}_q^p, p \notin \mathscr{P}_q} D_p[k]\} && (24)\\ &= \max\{\max_{p \in \mathscr{P}_q} D_p[k], \max_{\mathscr{N}_c \in \mathscr{C}_q} \{\max_{p \in \mathscr{D}_c^p} D_p[k]\}\} && (25)\\ &= \max\{\max_{p \in \mathscr{P}_q} D_p[k], \max_{\mathscr{N}_c \in \mathscr{C}_q} B_1(\mathscr{N}_c)\} && (26) \end{aligned}$$

Suppose we have, at some point in the traversal, two points $p_0, p_1 \in \mathscr{D}_q^p$ for some node $\mathscr{N}_q$, with $D_{p_0}[k] = \infty$ and $D_{p_1}[k] < \infty$. This means there exist k points $\{p_r^1, \ldots, p_r^k\}$ in S_r such that $d(p_1, p_r^i) \leq D_{p_1}[k]$ for $i = \{1, \ldots, k\}$. Because p_0 and p_1 are both descendant points of $\mathscr{N}_q$, we can apply the triangle inequality to see that $d(p_0, p_1) \leq 2\lambda_q$. Therefore, $d(p_0, p_r^i) \leq D_{p_1}[k] + 2\lambda(\mathscr{N}_q)$ for $i = \{1, \ldots, k\}$. Using this observation we can construct an alternate bound function $B_2(\mathscr{N}_q)$:

$$B_2(\mathscr{N}_q) = \min_{p \in \mathscr{D}_q^p} D_p[k] + 2\lambda(\mathscr{N}_q) \tag{27}$$

This bound can, like $B_1(\mathscr{N}_q)$, be rearranged to provide a recursive definition. In addition, if $p_0 \in \mathscr{P}_q$ and $p_1 \in \mathscr{D}_q^p$, we can bound $d(p_0, p_1)$ more tightly with $\rho(\mathscr{N}_q) + \lambda(\mathscr{N}_q)$ instead of $2\lambda(\mathscr{N}_q)$. These observations yield

$$\begin{aligned} B_2(\mathscr{N}_q) = \min\Bigg\{ &\min_{p \in \mathscr{P}_q}(D_p[k] + \rho(\mathscr{N}_q) + \lambda(\mathscr{N}_q)), \\ &\min_{\mathscr{N}_c \in \mathscr{C}_q}(B_2(\mathscr{N}_c) + 2(\lambda(\mathscr{N}_q) - \lambda(\mathscr{N}_c))) \Bigg\}. \end{aligned} \tag{28}$$

Both $B_1(\mathscr{N}_q)$ and $B_2(\mathscr{N}_r)$ provide valid pruning rules. We can combine both to get a tighter pruning rule by taking the tighter of the two bounds. In addition, $B_1(\mathscr{N}_q) \geq B_1(\mathscr{N}_c)$

Algorithm 15 k-nearest-neighbors `Score()`

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination should be pruned

if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q)$ **then**
  return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$

return ∞

and $B_2(\mathscr{N}_q) \geq B_2(\mathscr{N}_c)$ for all $\mathscr{N}_c \in \mathscr{C}_q$. Therefore, we can prune $(\mathscr{N}_q, \mathscr{N}_r)$ if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \geq \min\{B_1(\text{parent}(\mathscr{N}_q)), B_2(\text{parent}(\mathscr{N}_q))\}$.

These observations are combined for a better bound:

$$
\begin{aligned}
B(\mathscr{N}_q) = \min \Bigg\{ & \max\{ \max_{p \in \mathscr{P}_q} D_p[k], \max_{\mathscr{N}_c \in \mathscr{C}_q} B(\mathscr{N}_c) \}, \\
& \min_{p \in \mathscr{P}_q} (D_p[k] + \rho(\mathscr{N}_q) + \lambda(\mathscr{N}_q)), \\
& \min_{\mathscr{N}_c \in \mathscr{C}_q} \Big(B(\mathscr{N}_c) + 2(\lambda(\mathscr{N}_q) - \lambda(\mathscr{N}_c)) \Big) \\
& B(\text{parent}(\mathscr{N}_q)) \Bigg\}.
\end{aligned}
\tag{29}
$$

As a result of this bound function being expressed recursively, previous bounds can be cached and used to calculate the bound $B(\mathscr{N}_q)$ quickly. We can use this to structure `Score()` as given in Algorithm 15.

7.1.2 Correctness proof

A correctness proof is straightforward.

Theorem 3. *Given two datasets $S_q \in \mathfrak{R}^{N \times D}$ and $S_r \in \mathfrak{R}^{M \times D}$, a value k such that $0 < k < M$, two arbitrary space trees $\mathscr{T}_q$ and $\mathscr{T}_r$ built on S_q and S_r respectively, and an initially empty lists D and N, then any arbitrary pruning dual-tree traversal which uses Algorithm 14 for its* `BaseCase()` *and Algorithm 15 for its* `Score()` *will result in the list N_{p_q} being populated with the k nearest neighbors in S_r for each point $p_q \in S_q$, and $D_{p_q}[i] = \|p_q - N_{p_q}[i]\| \; \forall \; 0 < i \leq k, \; p_q \in S_q$.*

Proof. The proof can be split into two parts; first, we can prove that `BaseCase()` (Algorithm 14) is correct for a dual-tree (non-pruning) traversal. Then, we can prove that `Score()` (Algorithm 15) does not prune any node combinations which could contain any improvements to the list D.

Denote the list of true k nearest neighbors in S_r for $p_q \in S_q$ as $N^*_{p_q}$ and the corresponding distances as $D^*_{p_q}$. The `BaseCase()` implementation given in Algorithm 14 stores the k nearest neighbors and distances in S_r for each point in S_q. In a dual-tree traversal, `BaseCase()` is not called with any reference points that are not in S_r. Thus, it is clear that for N_{p_q} to be correct for each $p_q \in S_q$, then `BaseCase()` must be called with at least each combination of p_q with each of the k elements in $N^*_{p_q}$.

First, consider that in a dual-tree non-pruning traversal, each possible combination of nodes in $\mathscr{T}_q$ and $\mathscr{T}_r$ are visited. This follows from the definition. Also by definition, at each combination $(\mathscr{N}_q, \mathscr{N}_r)$, `BaseCase()` is called on each possible combination of points in $\mathscr{N}_q$ and $\mathscr{N}_r$.

Now, suppose there exists, for some query point $p_q \in \mathscr{S}_q$, a point p^*_r not in the final list D_{p_q} such that $\|p_q - p^*_r\| < D_{p_q}[k]$. Algorithm 14 will never discard a candidate with distance less than the current value of $D_{p_q}[k]$, but it is clear that at all times during the traversal, $\|p_q - p^*_r\| < D_{p_q}[k]$. Thus, p^*_r not being in the final list D_{p_q} implies that $p^*_r \notin \mathscr{S}_r$. This, with the fact that D_{p_q} is an ordered tuple list, shows that Algorithm 14 used in a dual-tree non-pruning traversal produces the correct results for k-nearest-neighbors. Note that each call `BaseCase(`p_q`,` p_r`)` with $p_r \notin D_{p_q}$ was unnecessary.

Next, consider the pruning rule given in Algorithm 15. A node combination is only pruned, according to the algorithm, if $d_{min}(\mathscr{N}_q, \mathscr{N}_r) \geq B(\mathscr{N}_q) \ \forall \ p_i \in \mathscr{D}^p_q$. $B(\mathscr{N}_q)$ is the minimum of four other bound functions. Each of those four bound functions was devised in such a way that at any point in the traversal, no node combination $(\mathscr{N}_q, \mathscr{N}_r)$ which could contain a point combination (p_q, p_r) where $\|p_q - p_r\| < D_{p_q}[k]$ is pruned. Because $D^*_{p_q}[k] \leq D_{p_q}[k]$ at all times during the traversal, then no point combination (p_q, p_r) where $p_r \in N^*_{p_q}$

Algorithm 16 `AllNN`($\mathscr{N}_q$, $\mathscr{N}_r$) [64]

if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \geq \delta_q^{nn}$, **then return**
if $\mathscr{N}_q$ is leaf **and** $\mathscr{N}_r$ is leaf **then**
  for all $p_q \in \mathscr{P}_q, p_r \in \mathscr{P}_r$ **do**
    $d_{qr} \leftarrow \|p_q - p_r\|$.
    if $d_{qr} < D_{p_q}$ **then** $D_{p_q} = d_{qr}$; $N_{p_q} = p_r$
    if $d_{qr} < \delta_q^{nn}$ **then** $\delta_q^{nn} \leftarrow d_{qr}$
`AllNN`($\mathscr{N}_q$.left, closer-of($\mathscr{N}_r$.left, $\mathscr{N}_r$.right))
`AllNN`($\mathscr{N}_q$.left, farther-of($\mathscr{N}_r$.left, $\mathscr{N}_r$.right))
`AllNN`($\mathscr{N}_q$.right, closer-of($\mathscr{N}_r$.left, $\mathscr{N}_r$.right))
`AllNN`($\mathscr{N}_q$.right, farther-of($\mathscr{N}_r$.left, $\mathscr{N}_r$.right))
$\delta_q^{nn} = \min(\delta_q^{nn}, \max(\delta_{q.\text{left}}^{nn}, \delta_{q.\text{right}}^{nn}))$

is ever pruned. Thus, `BaseCase()` is called with at least every point combination (p_q, p_r) where $p_r \in N_{p_q}^*$ for all $p_q \in S_q$. Therefore, at the end of the traversal, $N = N^*$ and $D = D^*$, so the theorem holds. □

7.1.3 Specialization to existing k-NN algorithms

This algorithm is a generalization of the standard kd-tree k-NN search, which uses a pruning dual-tree depth-first traversal. The archetypal algorithm for all-nearest neighbor search (k-nearest neighbor search with $k = 1$) given for kd-trees in Alex Gray's Ph.D. thesis [64] is shown here in Algorithm 16 with converted notation. δ_q^{nn} is the bound for a node $\mathscr{N}_q$ and is initialized to ∞; D_{p_q} represents the nearest distance for a query point p_q, and N_{p_q} represents the nearest neighbor for a query point p_q. $\mathscr{N}_q$.left represents the left child of $\mathscr{N}_q$ and is defined to be $\mathscr{N}_q$ if $\mathscr{N}_q$ has no children; $\mathscr{N}_q$.right is similarly defined.

The structure of the algorithm matches Algorithm 7; it is a dual-tree depth-first recursion. Because this is a depth-first recursion, $\delta_q^{nn} = \infty$ for a node $\mathscr{N}_q$ if no descendants of $\mathscr{N}_q$ have been recursed into. Otherwise, δ_q^{nn} is the maximum of D_{p_q} for all $p_q \in \mathscr{D}_q^p$. That is, $\delta_q^{nn} = B_1(\mathscr{N}_q)$. Thus, the comparison in the first line of Algorithm 16 is equivalent to Algorithm 15 with $B_1(\mathscr{N}_q)$ instead of $B(\mathscr{N}_q)$.

This algorithm is also a generalization of the standard cover tree k-NN search [57]. The cover tree search is a pruning dual-tree traversal where the query tree is traversed depth-first

while the reference tree is simultaneously traversed breadth-first. The pruning rule (after simple adaptation to the k-nearest-neighbor search problem instead of the nearest-neighbor search problem) is equivalent to

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \geq D_{p_q}[k] + \lambda_q \tag{30}$$

where p_q is the point contained in $\mathscr{N}_q$ (remember, each node of a cover tree contains one point). This is equivalent to $B_2(\mathscr{N}_q)$ because $\rho(\mathscr{N}_q) = 0$ for cover trees. The transformation from the algorithm given by Beygelzimer et al. [57] to our representation is clearer when considering the tree-independent form of the cover tree traversal (Algorithm 8) and also in the k-nearest neighbor search implementation of **mlpack** [159]; this implementation follows the API laid out in Chapter 4.

Specific algorithms for ball trees, metric trees, VP trees, octrees, and other space trees are trivial to create using the `BaseCase()` and `Score()` implementation given here (and in **mlpack**). Note also that this implementation will work in any metric space.

An extension to k-furthest neighbor search is straightforward. The bound function must be 'inverted' by changing 'max' to 'min' (and vice versa); in addition, the distances $D_{p_q}[i]$ must be initialized to 0 instead of ∞, and the lists D and N must be sorted by descending distance instead of ascending distance. Lastly, the comparison $d < D_{p_q}[k]$ must be changed to $d > D_{p_q}[k]$. With these simple changes, we have easily solved an entirely different problem using our meta-algorithm. An implementation using our meta-algorithm for both kd-trees and cover trees is also available in **mlpack**.

7.1.4 Runtime bounds

We now consider the running time of the algorithm, but with two important specializations for simplicity:

1. $k = 1$; we only show bounds for nearest neighbor search, not generalized k-nearest neighbor search. The bounds we present can be adapted but the exposition is more

complex.

2. The tree type is the cover tree and the traversal is the standard cover tree pruning traversal (see Section 5.2).

Because we are using the cover tree, we will simplify our bound function by bounding the bound function:

$$B(\mathscr{N}_q) \leq D[p_q] + \lambda_q \tag{31}$$

and then we may also bound λ_q (because we have restricted the tree type to cover trees) to see

$$B(\mathscr{N}_q) \leq D[p_q] + 2^{s_q+1} \tag{32}$$

wherein s_q is the scale of the query node $\mathscr{N}_q$[2].

Now, using the expansion constant c_r of the reference set S_r and the expansion constant c_q of the query set S_q, and defining

$$c_{qr} = \max\left(\left(\max_{p_q \in S_q} c'_r\right), c_r\right), \tag{33}$$

where c'_r is the expansion constant of the set $S_r \cup \{p_q\}$.

Theorem 4. *Using cover trees, the standard cover tree pruning dual-tree traversal, and the nearest neighbor search* `BaseCase()` *and* `Score()` *as given in Algorithms 14 and 15 with* $k = 1$*, respectively, and also given a reference set* S_r *with expansion constant* c_r*, and a query set* S_q*, the running time of the algorithm is bounded by* $O(c_r^4 c_{qr}^5 (N + i_t(\mathscr{T}_q) + \theta))$ *with* $i_t(\mathscr{T}_q$ *and* θ *defined as in Definition 10 and Lemma 4, respectively.*

Proof. The running time of `BaseCase()` and `Score()` are clearly $O(1)$. Due to Theorem 1, we therefore know that the runtime of the algorithm is bounded by $O(c_r^4 |R^*|(N + i_t(\mathscr{T}_q) + \theta))$. Thus, the only thing that remains is to bound the maximum size of the reference set, $|R^*|$.

[2]If the term 'scale' is unfamiliar, refer to the extensive discussion of cover trees in Sections 5.1.1 and 5.2.

Assume that when R^* is encountered, the maximum reference scale is $s_r^{\max}$ and the query node is $\mathcal{N}_q$. Every node $\mathcal{N}_r \in R^*$ satisfies the property enforced in line 10 that $d_{\min}(\mathcal{N}_q, \mathcal{N}_r) \leq B(\mathcal{N}_q)$. Using the definition of $d_{\min}(\cdot,\cdot)$ and $B(\cdot)$, we expand the equation. Note that p_q is the point held in $\mathcal{N}_q$ and p_r is the point held in $\mathcal{N}_r$. Also, take $\hat{p}_r$ to be the current nearest neighbor candidate for p_q; that is, $D[p_q] = d(p_q, \hat{p}_r)$ and $N[p_q] = \hat{p}_r$. Then,

$$\begin{aligned} d_{\min}(\mathcal{N}_q, \mathcal{N}_r) &\leq B(\mathcal{N}_q) && (34)\\ d(p_q, p_r) &\leq d(p_q, \hat{p}_r) + 2^{s_q+1} + 2^{s_r+1} + 2^{s_q+1} && (35)\\ &\leq d(p_q, \hat{p}_r) + 2(2^{s_r^{\max}+1}) && (36) \end{aligned}$$

where the last step follows because $s_q + 1 \leq s_r^{\max}$ and $s_r \leq s_r^{\max}$. Define the set of points P as the points held in each node in R^* (that is, $P = \{p_r \in \mathscr{P}(\mathcal{N}_r) : \mathcal{N}_r \in R^*\}$). Then, we can write

$$P \subseteq B_{S_r}(p_q, d(p_q, \hat{p}_r) + 2(2^{s_r^{\max}+1})). \tag{37}$$

Suppose that the true nearest neighbor is p_r^* and $d(p_q, p_r^*) > 2^{s_r^{\max}+1}$. Then, p_r^* must be held as a descendant point of some node in R^* which holds some point $\tilde{p}_r$. Using the triangle inequality,

$$d(p_q, \hat{p}_r) \leq d(p_q, \tilde{p}_r) \leq d(p_q, p_r^*) + d(\tilde{p}_r, p_r^*) \leq d(p_q, p_r^*) + 2^{s_r^{\max}+1}. \tag{38}$$

This gives that $P \subseteq B_{S_r \cup \{p_q\}}(p_q, d(p_q, p_r^*) + 3(2^{s_r^{\max}+1}))$. The previous step is necessary: to apply the definition of the expansion constant, the ball must be centered at a point in the set; now, the center (p_q) is part of the set.

$$\begin{aligned} |B_{S_r \cup \{p_q\}}(p_q, d(p_q, p_r^*) + 3(2^{s_r^{\max}+1}))| &\leq |B_{S_r \cup \{p_q\}}(p_q, 4d(p_q, p_r^*))| && (39)\\ &\leq c_{qr}^3 |B_{S_r \cup \{p_q\}}(p_q, d(p_q, p_r^*)/2)| && (40) \end{aligned}$$

which follows because the expansion constant of the set $S_r \cup \{p_q\}$ is bounded above by c_{qr}. Next, we know that p_r^* is the closest point to p_q in $S_r \cup \{p_q\}$; thus, there cannot exist a point $p_r' \neq p_q \in S_r \cup \{p_q\}$ such that $p_r' \in B_{S_{qr}}(p_q, d(p_q, p_r^*)/2)$ because that would imply that $d(p_q, p_r') < d(p_q, p_r^*)$, which is a contradiction. Thus, the only point in the ball is p_q, and we have that $|B_{S_r \cup \{p_q\}}(p_q, d(p_q, p_r^*)/2)| = 1$, giving the result that $|R| \leq c_{qr}^3$ in this case.

The other case is when $d(p_q, p_r^*) \leq 2^{s_r^{\max}+1}$, which means that $d(p_q, \hat{p}_r) \leq 2^{s_r^{\max}+2}$. Note that $P \in C_{s_r^{\max}}$, and therefore

$$P \subseteq B_{S_r}(p_q, d(p_q, p_r^*) + 3(2^{s_r^{\max}+1})) \cap C_{s_r^{\max}} \tag{41}$$

$$\subseteq B_{S_r}(p_q, 4(2^{s_r^{\max}+1})) \cap C_{s_r^{\max}}. \tag{42}$$

Every point in $C_{s_r^{\max}}$ is separated by at least $2^{s_r^{\max}}$. Using Lemma 1 with $\delta = 2^{s_r^{\max}}$ and $\rho = 8$ yields that $|P| \leq c_r^5$. This gives the result, because $c_r^5 \leq c_{qr}^5$. □

In the monochromatic case where $S_q = S_r$, the bound is $O(c^9(N + i_t(\mathscr{T}))$ because $c = c_r = c_{qr}$ and $\theta = 0$. For well-behaved trees where $i_t(\mathscr{T}_q)$ is linear or sublinear in N, this represents the current tightest worst-case runtime bound for nearest neighbor search.

7.2 Range search

Range search is another popular neighbor searching problem related to k-nearest neighbor search. In addition to being a fairly standard machine learning task, it has numerous uses in applications such as databases and geographic information systems (GIS). A treatise on the history of the problem and solutions is given by Agarwal & Erickson [160]. The problem is:

Given query and reference datasets S_q, S_r and a range $[l, u]$, for each point $p_q \in S_q$, find all points in S_r such that $l \leq \|p_q - p_r\| \leq u$. Refer to the list of neighbors for each query point p_q as $S[p_q]$. This list is not sorted in any particular order, and at initialization time, it is empty.

Algorithm 17 Range search `BaseCase()`

Input: query point p_q, reference point p_r, range sets $N[p_q]$ and range $[l, u]$
Output: distance d between p_q and p_r
if $d(p_q, p_r) \in [r_{\min}, r_{\max}]$ **and** `BaseCase(`p_q`,` p_r`)` not yet called **then**
    $S[p_q] \leftarrow S[p_q] \cup \{p_r\}$
return d

Algorithm 18 Range search `Score()`

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination should be pruned
if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \in [l, u]$ or $d_{\max}(\mathscr{N}_q, \mathscr{N}_r) \in [l, u]$ **then**
    return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
return ∞

In different settings, the problem of range search may not be stated identically; however, our results are easily adaptable. A closely related problem is range count, where instead of the set $S[p_q]$, only the size of the set $|S[p_q]|$ is desired for each query point p_q.

While range search is sometimes mentioned in the context of dual-tree algorithms [61], the focus is usually on k-nearest neighbor search. As a result, I cannot find any explicitly published dual-tree algorithms to generalize; however, a single-tree algorithm was proposed by Bentley and Friedman [37]. Therefore, we will develop a novel tree-independent dual-tree algorithm for range search, which is easily adaptible to range count.

7.2.1 A tree-independent dual-tree algorithm

Range search turns out to be far simpler than k-nearest neighbor search, mainly because there is no complex bounding function $B(\mathscr{N}_q)$ for pruning. Pruning is only necessary when we can determine that for a node combination $(\mathscr{N}_q, \mathscr{N}_r)$, no descendant points of $\mathscr{N}_r$ could possibly be in the range $[l, u]$ for any descendant point of $\mathscr{N}_q$. This also means that recursion order does not matter for range search.

`BaseCase()` and `Score()` functions are given in Algorithms 17 and 18. Algorithm 17, the `BaseCase()` function, merely needs to add a reference point p_r to the range set $S[p_q]$ if

Algorithm 19 More complicated `Score()` for range search.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination should be pruned

if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \in [l, u]$ or $d_{\max}(\mathscr{N}_q, \mathscr{N}_r) \in [l, u]$ **then**
  return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
else if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) \geq l$ and $d_{\max}(\mathscr{N}_q, \mathscr{N}_r) \leq u$ **then**
  for all $p_q \in \mathscr{D}_q^p$ **do**
    $S[p_q] \leftarrow S[p_q] \cup \mathscr{D}_r^p$
return ∞

$d(p_q, p_r)$ lies in the desired range. For pruning, observe that a node combination $(\mathscr{N}_q, \mathscr{N}_r)$ can be pruned if no descendant reference point of $\mathscr{N}_r$ can possibly fall within the desired range of any descendant query point of $\mathscr{N}_q$. This is straightforward to formalize; we may prune if

$$[d_{\min}(\mathscr{N}_q, \mathscr{N}_r), d_{\max}(\mathscr{N}_q, \mathscr{N}_r)] \cup [l, u] \neq \emptyset. \tag{43}$$

The given `Score()` algorithm expresses this in a more simple manner.

However, there is another pruning possibility that `Score()` in Algorithm 18 does not exploit: if $[d_{\min}(\mathscr{N}_q, \mathscr{N}_r), d_{\max}(\mathscr{N}_q, \mathscr{N}_r)]$ falls completely within the desired range $[l, u]$, then *every* descendant reference point of $\mathscr{N}_r$ must fall into the result set for *every* descendant query point of $\mathscr{N}_q$. A more complex `Score()` algorithm is given in Algorithm 19. This algorithm is implemented in **mlpack**.

7.2.2 Runtime bound

We turn our attention to bounding the running time of range search (and range count). Until the recent work which is presented in this section [135], there was no existing bound for range search which was better than the bound for brute-force range search ($O(NM)$ for a query set of size M and a reference set of size N). It turns out that the simpler `Score()` (Algorithm 18) is sufficient for a runtime bound, so in this section we will consider that implementation, as opposed to the more complex `Score()` of Algorithm 19. The bound

will apply for both algorithms, though, since the more complex `Score()` is easily shown to only ever do less work than the simpler `Score()`.

In order to bound the running time of dual-tree range search, we require better notions for understanding the difficulty of the problem. Observe that if the range is sufficiently large, then for every query point p_q, $S[p_q] = S_r$. Clearly, for $S_q \sim S_r \sim O(N)$, this cannot be solved in anything less than quadratic time simply due to the time required to fill each output array $S[p_q]$. Define the maximum result size for a given query set S_q, reference set S_r, and range $[l, u]$ as

$$|S_{\max}| = \max_{p_q \in S_q} |S[p_q]|. \tag{44}$$

Small $|S_{\max}|$ implies an easy problem; large $|S_{\max}|$ implies a difficult problem. For bounding the running time of range search, we require one more notion of difficulty, related to how $|S_{\max}|$ changes due to changes in the range $[l, u]$.

Definition 12. *For a range search problem with query set S_q, reference set S_r, range $[l, u]$, and results $S[p_q]$ for each query point p_q given as*

$$S[p_q] = \{p_r : p_r \in S_r, l \leq d(p_q, p_r) \leq u\}, \tag{45}$$

define the α-expansion of the range set $S[p_q]$ as the slightly larger set

$$S^{\alpha}[p_q] = \{p_r : p_r \in S_r, (1 - \alpha)l \leq d(p_q, p_r) \leq (1 + \alpha)u\}. \tag{46}$$

When the α-expansion of the set $S_{\max}$ is approximately the same size as $S_{\max}$, then the problem would not be significantly more difficult if the range $[l, u]$ was increased slightly. Using these notions, then, we may now bound the running time of range search.

Theorem 5. *Given a reference set S_r of size N with expansion constant c_r, and a query set S_q of size $O(N)$, a search range of $[l, u]$, and using the range search `BaseCase()`*

and `Score()` *as given in Algorithms 17 and 18, respectively, with the standard cover tree pruning dual-tree traversal as given in Algorithm 8, and also assuming that for some* $\alpha > 0$*,*

$$|S^{\alpha}[p_q] \setminus S[p_q]| \leq C \quad \forall\ p_q \in S_q, \tag{47}$$

the running time of range search or range count is bounded by

$$O\left(c_r^4 \max\left(c_r^{4+\beta}, |S_{\max}| + C\right)(N + i_t(\mathscr{N}_q) + \theta)\right) \tag{48}$$

with θ *defined as in Lemma 4,* $\beta = \lceil \log_2(1 + \alpha^{-1}) \rceil$*, and* $S_{\max}$ *as defined in Equation 44.*

Proof. Both `BaseCase()` (Algorithm 17) and `Score()` (Algorithm 18) take $O(1)$ time. Therefore, using Lemma 1, we know that the runtime of the algorithm is bounded by $O(c_r^4 |R^*|(N + i_t(\mathscr{N}_q) + \theta))$. As with the previous proofs, then, our only task is to bound the maximum size of the reference set, $|R^*|$.

By the pruning rule, for a query node $\mathscr{N}_q$, the reference set R^* is made up of reference nodes $\mathscr{N}_r$ that are within a margin of $2^{s_q+1} + 2^{s_r+1} \leq 2^{s_r^{\max}+2}$ of the range $[l, u]$. Given that p_r is the point in $\mathscr{N}_r$,

$$p_r \in \left(B_{S_r}(p_q, u + 2^{s_r^{\max}+2}) \cap C_{s_r^{\max}}\right) \setminus \left(B_{S_r}(p_q, l - 2^{s_r^{\max}+2}) \cap C_{s_r^{\max}}\right). \tag{49}$$

A bound on the number of elements in this set is a bound on $|R^*|$. First, consider the case where $u \leq \alpha^{-1} 2^{s_r^{\max}+2}$. Ignoring the smaller ball, take $\delta = 2^{s_r^{\max}}$ and $\rho = 4(1 + \alpha^{-1})$ and apply Lemma 1 to produce the bound

$$|R^*| \leq c_r^{4+\lceil \log_2(1+\alpha^{-1}) \rceil}. \tag{50}$$

Now, consider the other case: $u > \alpha^{-1} 2^{s_r^{\max}+1}$. This means

$$B_{S_r}(p_q, u + 2^{s_r^{\max}+1}) \setminus B_{S_r}(p_q, l - 2^{s_r^{\max}+1}) \subseteq B_{S_r}(p_q, (1 + \alpha)u) \setminus B_{S_r}(p_q, (1 - \alpha)l). \tag{51}$$

This set is necessarily a subset of $S^{\alpha}[p_q]$; by assumption, the number of points in this set is bounded above by $|S_{\max}| + C$. We may then conclude that $|R^*| \leq |S_{\max}| + C$. By taking the maximum of the sizes of $|R^*|$ in both cases above, we obtain the statement of the theorem. □

This bound displays both the expected dependence on c_r and $|S_{\max}|$. As the largest range set $S_{\max}$ increases in size (with the worst case being $S_{\max} \sim N$), the runtime degenerates to quadratic. But for adequately small $S_{\max}$ the runtime is instead dependent on c_r and the parameter C of the α-expansion of $S_{\max}$. This situation leads to a simplification.

Corollary 2. *For sufficiently small $|S_{\max}|$ and sufficiently small C, the runtime of range search under the conditions of Theorem 5 simplifies to*

$$O(c_r^{8+\beta}(N + i_t(\mathscr{N}_q) + \theta)). \tag{52}$$

In this setting we can more easily consider the relation of the running time to α. Consider $\alpha = (1/3)$; this yields a running time of $O(c^8(N + \theta))$. $\alpha = (1/7)$ yields $O(c^9(N + i_t(\mathscr{N}_q + \theta))$, $\alpha = (1/15)$ yields $O(c^{10}(N + i_t(\mathscr{N}_q) + \theta))$, and so forth. As α gets smaller, the exponent on c gets larger, and diverges as $\alpha \to 0$.

For reasonable runtime it is necessary that the α-expansion of $S_{\max}$ be bounded. This is because the dual-tree recursion must retain reference nodes which may contain descendants in the range set $S[p_q]$ for some query p_q. The parameter C of the α-expansion allows us to bound the number of reference nodes of this type, and if α increases but C remains small enough that Corollary 2 applies, then we are able to obtain tighter running bounds.

It is worth reiterating that the bound here depends only on the pruning rule of Algorithm 18, not the more complex pruning rule of Algorithm 19; thus, our bound is potentially somewhat looser than it could be. Unfortunately, the expansion constant does not make

working with slices of balls easy, and therefore the exact route to a tighter bound with the more complex pruning rule is unclear.

7.3 Kernel density estimation

Much work has been produced regarding the use of dual-tree algorithms for kernel density estimation (KDE), including by Gray & Moore [61, 65] and later by Lee et al. [73, 75]. Kernel density estimation is an important machine learning task with a vast range of applications, from signal processing to econometrics.

Given a reference set S_r, a query point p_q, and a kernel $\mathcal{K}(\cdot, \cdot)$, the true kernel density estimate for a query point p_q is given as

$$f^*(p_q) = \sum_{p_r \in S_r} \mathcal{K}(p_q, p_r). \tag{53}$$

Often, the kernel $\mathcal{K}(\cdot, \cdot)$ is shift-invariant; that is, it is just a function of the distance between two points:

$$\mathcal{K}(p_q, p_r) := \mathcal{K}(\|p_q - p_r\|). \tag{54}$$

Some common examples of kernels include the Gaussian, Epanechnikov, Laplacian, exponential, and hyperbolic tangent kernels.

In the case of an infinite-tailed shift-invariant kernel $\mathcal{K}(\cdot, \cdot)$, the exact computation cannot be accelerated; thus, attention has turned towards tractable approximation schemes. Two simple schemes for the approximation of $f^*(p_q)$ are well-known: *absolute value approximation* and *relative value approximation*. Absolute value approximation requires that each density estimate $f(p_q)$ is within ϵ of the true estimate $f^*(p_q)$:

$$|f(p_q) - f^*(p_q)| < \epsilon \quad \forall p_q \in S_q. \tag{55}$$

Relative value approximation is a more flexible approximation scheme; given some parameter ϵ, the requirement is that each density estimate is within a relative tolerance of

$f^*(p_q)$:

$$\frac{|f(p_q) - f^*(p_q)|}{|f^*(p_q)|} < \epsilon \quad \forall p_q \in S_q. \tag{56}$$

Kernel density estimation is related to the well-studied problem of kernel summation, which can also be solved with dual-tree algorithms [74, 75]. In both of those problems, regardless of the approximation scheme, simple geometric observations can be made to accelerate computation: when $\mathcal{K}(\cdot, \cdot)$ is shift-invariant, faraway points have very small kernel evaluations. Thus, trees can be built on S_q and S_r, and node combinations can be pruned when the nodes are far apart while still obeying the error bounds.

In the following two subsections, we will show two simple dual-tree algorithms for both absolute-value and relative-value approximate kernel density estimation. We additionally restrict ourselves to the standard kernel density estimation assumptions of a shift-invariant kernel $\mathcal{K}(p_q, p_r) = \mathcal{K}(\|p_q - p_r\|)$ which is monotonically decreasing and non-negative. These dual-tree algorithms are useful when density estimates are required for not just a single query point p_q but instead an entire query set S_q. Then, we will analyze the running time of each algorithm.

7.3.1 Dual-tree algorithm for absolute-value approximation

A tree-independent algorithm for solving approximate kernel density estimation with absolute value approximation under the previous assumptions on the kernel is given as a `BaseCase()` function in Algorithm 20 and a `Score()` function in Algorithm 21. The list f_p holds partial kernel density estimates for each query point, and the list f_n holds partial kernel density estimates for each query node. At the beginning of the dual-tree traversal, the lists f_p and f_n, which are both of size $O(N)$, are each initialized to 0. As the traversal proceeds, node combinations are pruned if the difference between the maximum kernel value $\mathcal{K}(d_{\min}(\mathcal{N}_q, \mathcal{N}_r))$ and the minimum kernel value $\mathcal{K}(d_{\max}(\mathcal{N}_q, \mathcal{N}_r))$ is sufficiently small (line 3). If the node combination can be pruned, then the partial node estimate is updated

Algorithm 20 Approximate kernel density estimation `BaseCase()`

1: **Input:** query point p_q, reference point p_r, list of kernel point estimates $\hat{f}_p$
2: **Output:** kernel value $\mathcal{K}(p_q, p_r)$
3: $f_p(p_q) \leftarrow f_p(p_q) + \mathcal{K}(p_q, p_r)$
4: **return** $\mathcal{K}(p_q, p_r)$

Algorithm 21 Absolute-value approximate kernel density estimation `Score()`

1: **Input:** query node $\mathcal{N}_q$, reference node $\mathcal{N}_r$, list of node kernel estimates $\hat{f}_n$
2: **Output:** a score for the node combination $(\mathcal{N}_q, \mathcal{N}_r)$, or ∞ if the combination should be pruned
3: **if** $\mathcal{K}(d_{\min}(\mathcal{N}_q, \mathcal{N}_r)) - \mathcal{K}(d_{\max}(\mathcal{N}_q, \mathcal{N}_r)) < \epsilon$ **then**
4: $\quad f_n(\mathcal{N}_q) \leftarrow f_n(\mathcal{N}_q) + |\mathscr{D}^p(\mathcal{N}_r)| \left(\mathcal{K}(d_{\min}(\mathcal{N}_q, \mathcal{N}_r)) + \mathcal{K}(d_{\max}(\mathcal{N}_q, \mathcal{N}_r))\right) / 2$
5: $\quad$ **return** ∞
6: **return** $\mathcal{K}(d_{\min}(\mathcal{N}_q, \mathcal{N}_r)) - \mathcal{K}(d_{\max}(\mathcal{N}_q, \mathcal{N}_r))$

(line 4). When node combinations cannot be pruned, `BaseCase()` may be called, which simply updates the partial point estimate with the exact kernel evaluation (line 3).

After the dual-tree traversal, the actual kernel density estimates f must be extracted. This can be done by traversing the query tree and calculating $f(p_q) = f_p(p_q) + \sum_{\mathcal{N}_i \in T} f_n(\mathcal{N}_i)$, where T is the set of nodes in $\mathscr{T}_q$ that have p_q as a descendant. Each query node needs to be visited only once to perform this calculation; it may therefore be accomplished in $O(N)$ time.

Note that this version is far simpler than other dual-tree algorithms that have been proposed for approximate kernel density estimation (see, for instance, Gray's algorithm [65]); however, this version is sufficient for our runtime analysis. Real-world implementations tend to be far more complex.

Proving correct functionality of kernel density estimation is simple. In Algorithm 21, each pruned subtree introduces a maximum of $\epsilon(|\mathscr{D}_r^p|/|S_r|)$ error into the density estimate. Clearly, we cannot prune more than $|S_r|$ reference points, giving a maximum of $\epsilon(|S_r|/|S_r|) = \epsilon$ approximation error. In addition, for every reference point not pruned, we correctly add its contribution to the density estimate in Algorithm 20. Thus, our algorithm

produces approximate kernel density estimates within the bounds specified by the error parameter ϵ. Often, the actual empirical error for any query point will be significantly smaller than ϵ.

7.3.2 Absolute-value approximate KDE runtime bounds

If we place additional restrictions on the dual-tree algorithm, we may use adaptive algorithm analysis techniques to bound the running time. The restrictions are the usual restrictions: we will use the cover tree (Section 5.1.1) and standard cover tree traversal (Algorithm 8). In addition, we will require that the shift-invariant kernel $\mathcal{K}(\cdot, \cdot)$ satisfies the following properties: there exists some bandwidth h such that $\mathcal{K}(d)$ must be concave for $d \in [0, h]$ and convex for $d \in [h, \infty)$. This assumption implies that the magnitude of the derivative $|\mathcal{K}'(d)|$ is maximized at $d = h$. This assumption is not restrictive; most standard kernels fall into this class, including the Gaussian, exponential, and Epanechnikov kernels.

Theorem 6. *Assume that $\mathcal{K}(\cdot, \cdot)$ is a kernel satisfying the assumptions above. Then, given a query set S_q and a reference set S_r with expansion constant c_r, and using the approximate kernel density estimation* `BaseCase()` *and* `Score()` *as given in Algorithms 20 and 21, respectively, with the traversal given in Algorithm 8, the running time of approximate kernel density estimation for some error parameter ϵ is bounded by $O(c_r^{8+\lceil \log_2 \zeta \rceil}(N + i_t(\mathscr{T}_q) + \theta))$ with $\zeta = -\mathcal{K}'(h)\mathcal{K}^{-1}(\epsilon)\epsilon^{-1}$, $i_t(\mathscr{T}_q)$ defined as in Definition 10, and θ defined as in Lemma 4.*

Proof. It is clear that `BaseCase()` and `Score()` both take $O(1)$ time, so Theorem 1 implies the total runtime of the dual-tree algorithm is bounded by $O(c_r^4 |R^*|(N + i_t(\mathscr{T}_q) + \theta))$. Thus, we will bound $|R^*|$ using techniques related to those used by Ram et al. [133]. The bounding of $|R^*|$ is split into two sections: first, we show that when the scale $s_r^{\max}$ is small enough, R^* is empty. Second, we bound R^* when $s_r^{\max}$ is larger.

The `Score()` function is such that any node in R^* for a given query node $\mathscr{N}_q$ obeys

$$\mathcal{K}(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)) - \mathcal{K}(d_{\max}(\mathscr{N}_q, \mathscr{N}_r)) \geq \epsilon. \tag{57}$$

Thus, we are interested in the maximum possible value $\mathcal{K}(a) - \mathcal{K}(b)$ for a fixed value of $b - a > 0$. Due to our assumptions, the maximum value of $\mathcal{K}'(\cdot)$ is $\mathcal{K}'(h)$; therefore, the maximum possible value of $\mathcal{K}(a) - \mathcal{K}(b)$ is when the interval $[a, b]$ is centered on h. This allows us to say that $\mathcal{K}(a) - \mathcal{K}(b) \leq \epsilon$ when $(b - a) \leq (-\epsilon/\mathcal{K}'(h))$. Note that

$$\begin{aligned} d_{\max}(\mathscr{N}_q, \mathscr{N}_r) - d_{\min}(\mathscr{N}_q, \mathscr{N}_r) &\leq d(p_q, p_r) + 2^{s_r^{\max}+1} - d(p_q, p_r) + 2^{s_r^{\max}+1} & (58) \\ &\leq 2^{s_r^{\max}+2}. & (59) \end{aligned}$$

Therefore, $R^* = \emptyset$ when $2^{s_r^{\max}+2} \leq -\epsilon/\mathcal{K}'(h)$, or when $s_r^{\max} \leq \log_2(-\epsilon/\mathcal{K}'(h)) - 2$. Consider, then, the case when $s_r^{\max} > \log_2(-\epsilon/\mathcal{K}'(h)) - 2$. Because of the pruning rule, for any $\mathscr{N}_r \in R^*$, $\mathcal{K}(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)) > \epsilon$; we may refactor this by applying definitions to show $d(p_q, p_r) < \mathcal{K}^{-1}(\epsilon) + 2^{s_r^{\max}+1}$. Therefore, bounding the number of points in the set $B_{S_r}(p_q, \mathcal{K}^{-1}(\epsilon) + 2^{s_r^{\max}+1}) \cap C_{s_r^{\max}}$ is sufficient to bound $|R^*|$. For notational convenience, define $\omega = (\mathcal{K}^{-1}(\epsilon)/2^{s_r^{\max}+1}) + 1$, and the statement may be more concisely written as $B_{S_r}(p_q, \omega 2^{s_r^{\max}+1}) \cap C_{s_r^{\max}}$.

Using Lemma 1 with $\delta = 2^{s_r^{\max}}$ and $\rho = 2\omega$ gives $|R^*| = c_r^{3+\lceil \log_2 \omega \rceil}$.

The value ω is maximized when $s_r^{\max}$ is minimized. Using the lower bound on $s_r^{\max}$, ω is bounded as $\omega = -2\mathcal{K}'(h)\mathcal{K}^{-1}(\epsilon)\epsilon^{-1}$. Finally, with $\zeta = -\mathcal{K}'(h)\mathcal{K}^{-1}(\epsilon)\epsilon^{-1}$, we are able to conclude that $|R^*| \leq c_r^{3+\lceil \log_2(2\zeta) \rceil} = c_r^{4+\lceil \log_2 \zeta \rceil}$. Therefore, the entire dual-tree traversal takes $O(c_r^{8+\lceil \log_2 \zeta \rceil}(N + \theta))$ time.

The postprocessing step to extract the estimates $f(\cdot)$ requires one traversal of the tree $\mathscr{T}_r$; the tree has $O(N)$ nodes, so this takes only $O(N)$ time. This is less than the runtime of the dual-tree traversal, so the runtime of the dual-tree traversal dominates the algorithm's runtime, and the theorem holds. □

The dependence on ϵ (through ζ) is expected: as $\epsilon \to 0$ and the search becomes exact, ζ

diverges both because ϵ^{-1} diverges and also because $\mathcal{K}^{-1}(\epsilon)$ diverges, and the runtime goes to the worst-case $O(N^2)$; exact kernel density estimation means no nodes can be pruned at all.

For the Gaussian kernel with bandwidth σ defined by $\mathcal{K}_g(d) = \exp(-d^2/(2\sigma^2))$, ζ does not depend on the kernel bandwidth; only the approximation parameter ϵ. For this kernel, $h = \sigma$ and therefore $-\mathcal{K}_g'(h) = \sigma^{-1}e^{-1/2}$. Additionally, $\mathcal{K}_g^{-1}(\epsilon) = \sigma\sqrt{2\ln(1/\epsilon)}$. This means that for the Gaussian kernel, $\zeta = \sqrt{(-2\ln\epsilon)/(e\epsilon^2)}$. Again, as $\epsilon \to 0$, the runtime diverges; however, note that there is no dependence on the kernel bandwidth σ. To demonstrate the relationship of runtime to ϵ, see that for a reasonably chosen $\epsilon = 0.05$, the runtime is approximately $O(c_r^{8.89}(N + \theta))$; for $\epsilon = 0.01$, the runtime is approximately $O(c_r^{11.52}(N + \theta))$. For very small $\epsilon = 0.00001$, the runtime is approximately $O(c_r^{22.15}(N + \theta))$.

Next, consider the exponential kernel: $\mathcal{K}_l(d) = \exp(-d/\sigma)$. For this kernel, $h = 0$ (that is, the kernel is always convex), so then $\mathcal{K}_l'(h) = \sigma^{-1}$. Simple algebraic manipulation gives $\mathcal{K}_l^{-1}(\epsilon) = -\sigma\ln\epsilon$, resulting in $\zeta = -\mathcal{K}_l'(h)\mathcal{K}_l^{-1}(\epsilon)\epsilon^{-1} = \epsilon^{-1}\ln\epsilon$. So both the exponential and Gaussian kernels do not exhibit dependence on the bandwidth.

To understand the lack of dependence on kernel bandwidth more intuitively, consider that as the kernel bandwidth increases, two things happen: *(a)* the reference set R becomes empty at larger scales, and *(b)* $\mathcal{K}^{-1}(\epsilon)$ grows, allowing less pruning at higher levels. These effects are opposite, and for the Gaussian and exponential kernels they cancel each other out, giving the same bound regardless of bandwidth.

7.3.3 Relative Value Approximation

It is straightforward to adapt the `Score()` function in Algorithm 21 to relative value approximation; the pruning condition only needs a little tweaking. `BaseCase()` can remain the same as in Algorithm 20.

First, we must establish a `Score()` function for relative value approximation. The difference between Equations 55 and 56 is the division by the term $|f^*(p_q)|$. But we can quickly bound $|f^*(p_q)|$:

Algorithm 22 Relative-value approximate kernel density estimation `Score()`

1: **Input:** query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$, list of node kernel estimates $\hat{f}_n$
2: **Output:** a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination should be pruned
3: **if** $\mathcal{K}(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)) - \mathcal{K}(d_{\max}(\mathscr{N}_q, \mathscr{N}_r)) < \epsilon\mathcal{K}^{\max}$ **then**
4: $\quad f_n(\mathscr{N}_q) \leftarrow f_n(\mathscr{N}_q) + |\mathscr{D}^p(\mathscr{N}_r)|\left(\mathcal{K}(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)) + \mathcal{K}(d_{\max}(\mathscr{N}_q, \mathscr{N}_r))\right) / 2$
5: $\quad$ **return** ∞
6: **return** $\mathcal{K}(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)) - \mathcal{K}(d_{\max}(\mathscr{N}_q, \mathscr{N}_r))$

$$|f^*(p_q)| \geq N\mathcal{K}\left(\max_{p_r \in S_r} d(p_q, p_r)\right). \tag{60}$$

This is clearly true: each point in S_r must contribute more than $\mathcal{K}(\max_{p_r \in S_r} d(p_q, p_r))$ to $f^*(p_q)$. Now, we may revise the relative approximation condition in Equation 56:

$$|f(p_q) - f^*(p_q)| \leq \epsilon\mathcal{K}^{\max} \tag{61}$$

where $\mathcal{K}^{\max}$ is lower bounded by $\mathcal{K}(\max_{p_r \in S_r} d(p_q, p_r))$. Assuming we have some estimate $\mathcal{K}^{\max}$, this allows us to create a `Score()` algorithm, given in Algorithm 22. An estimate $\mathcal{K}_{\max}$ may easily be obtained: one pass over both S_q and S_r can determine a bounding box (or ball) of the data, and then the diameter of the box (or sphere) can be used to produce an estimate $\mathcal{K}^{\max}$. This is only a strategy for a rough estimate; it is possible to produce tighter bounds by exploiting the already-built query and reference trees, as we will see in the upcoming proof.

7.3.4 Runtime bounds for relative value approximate KDE

Using the `Score()` function in Algorithm 22 and the runtime bound results for absolute value approximate kernel density estimation, we may prove linear runtime bounds for relative value approximate kernel density estimation.

Theorem 7. *Assume that $\mathcal{K}(\cdot, \cdot)$ is a kernel satisfying the same assumptions as Theorem 6. Then, given a query set S_q and a reference set S_r both of size $O(N)$, it is possible to perform*

relative value approximate kernel density estimation (satisfying the condition of Equation 56) in $O(N)$ time, assuming that the expansion constant c_r of S_r is not dependent on N.

Proof. It is easy to see that Theorem 6 may be adapted to the very slightly different `Score()` rule of Algorithm 22 while still providing an $O(N)$ bound. With that `Score()` function, the dual-tree algorithm will return relative-value approximate kernel density estimates satisfying Equation 56.

We now turn to the calculation of $\mathcal{K}^{\max}$. Given the cover trees $\mathcal{T}_q$ and $\mathcal{T}_r$ with root nodes $\mathcal{N}_r^R$ and $\mathcal{N}_r^R$, respectively, we may calculate a suitable $\mathcal{K}^{\max}$ value in constant time:

$$\mathcal{K}^{\max} = d_{\max}(\mathcal{N}_q^R, \mathcal{N}_r^R) = d(p_q^R, p_r^R) + 2^{s_q^R+1} + 2^{s_r^R+1}. \tag{62}$$

This proves the statement of the theorem. □

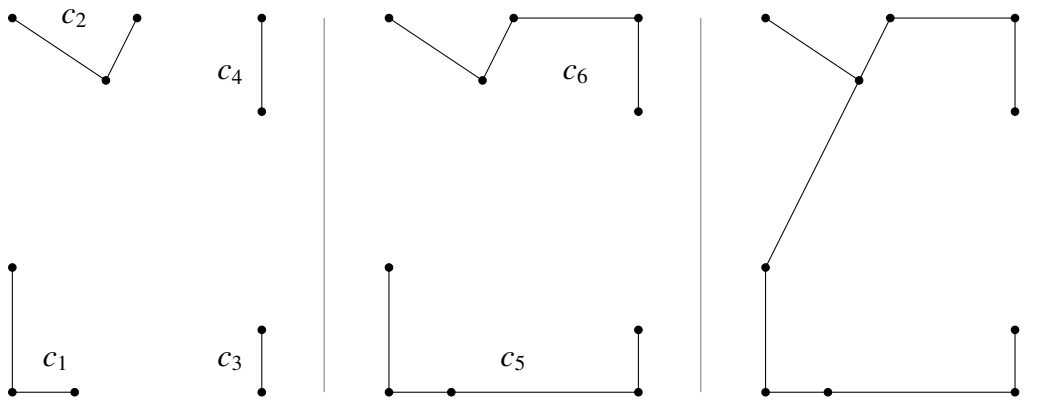
In this case, we have not shown tighter bounds because the algorithm we have proposed is not useful in practice. For an example of a better relative-value approximate kernel density estimation dual-tree algorithm, see the work of Gray [65].

With linear runtime bounds proved for both relative value approximate and absolute value approximate kernel density estimation, we move on to the next algorithm.

7.4 Minimum spanning tree calculation

Finding a Euclidean minimum spanning tree has been a relevant problem since Borůvka's algorithm was proposed in 1926. Recently, a dual-tree version of Borůvka's algorithm was developed [68] for *kd*-trees and cover trees. We unify these two algorithms and generalize to other types of space tree by formulating `BaseCase()` and `Score()` functions.

For a dataset $S_r \in \Re^{N \times D}$, Borůvka's algorithm connects each point to its nearest neighbor, giving many 'components'. For each component c, the nearest point in S_r to any point of c that is not part of c is found. The points are connected, combining those components. This process repeats until only one component—the minimum spanning tree—remains. Figure 32 shows a sample evolution of components.



(a) First stage. (b) Second stage. (c) Final MST.

Figure 32: Progression of Borůvka's algorithm.

During the algorithm, we maintain a list F made up of i components $F_i : \{E_i, V_i\}$ where E_i is the list of edges and V_i is the list of vertices in the component F_i (these are points in S_r). Each point in S_r belongs to only one F_i. At initialization, $|F| = |S_r|$ and $F_i = \{\emptyset, \{p_i\}\}$ for $i = \{1, \ldots, |S_r|\}$, where p_i is the i'th point in S_r. For $p \in S_r$ we define $F(p) = F_i$ if F_i is the component containing p. During the algorithm, we maintain $N(F_i)$ as the candidate nearest neighbor of component F_i and $p_c(F_i)$ as the point in component F_i nearest to $N(F_i)$. Then, $D(F_i) = \|p_c(F_i) - N(F_i)\|$. Remember that $F(N(F_i)) \neq F_i$.

To run Borůvka's algorithm with a space tree $\mathscr{T}_r$ built on the set of points S_r, a pruning dual-tree traversal is run with `BaseCase()` as Algorithm 23, `Score()` as Algorithm 24, $\mathscr{T}_r$ as *both* of the trees, and F as initialized before. Note that `Score()` uses $B(\mathscr{N}_q)$ from Section 7.1 with $k = 1$. Upon traversal completion, we have a list $N(F_i)$ of nearest neighbors of each

Algorithm 23 Borůvka's algorithm `BaseCase()`.

Input: query point p_q, reference point p_r, nearest candidate point $N(F(p_q))$ and distance $D(F(p_q))$
Output: distance d between p_q and p_r

if $p_q = p_r$ **then**
  return 0
if $F(p_q) \neq F(p_r)$ **and** $\|p_q - p_r\| < D(F(p_q))$ **then**
  $D(F(p_q)) \leftarrow \|p_q - p_r\|$
  $N(F(p_q)) \leftarrow p_r$; $\quad p_c(F(p_q)) \leftarrow p_q$
return $\|p_q - p_r\|$

Algorithm 24 Borůvka's algorithm `Score()`.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination should be pruned

if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q)$ **then**
  if $F(p_q) = F(p_r)\ \forall p_q \in \mathscr{D}_q^p, p_r \in \mathscr{D}_r^p$ **then**
    return ∞
  return $d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
return ∞

component F_i. The edge $(N(F_i), p_c(F_i))$ is added to F_i for each F_i. Then, any components in F with shared edges are merged into a new list F' where $|F'| < |F|$. The pruning dual-tree traversal is then run again with $F = F'$ and the traversal-merge process repeats until $|F| = 1$. When $|F| = 1$, then F_1 is the minimum spanning tree of S_r.

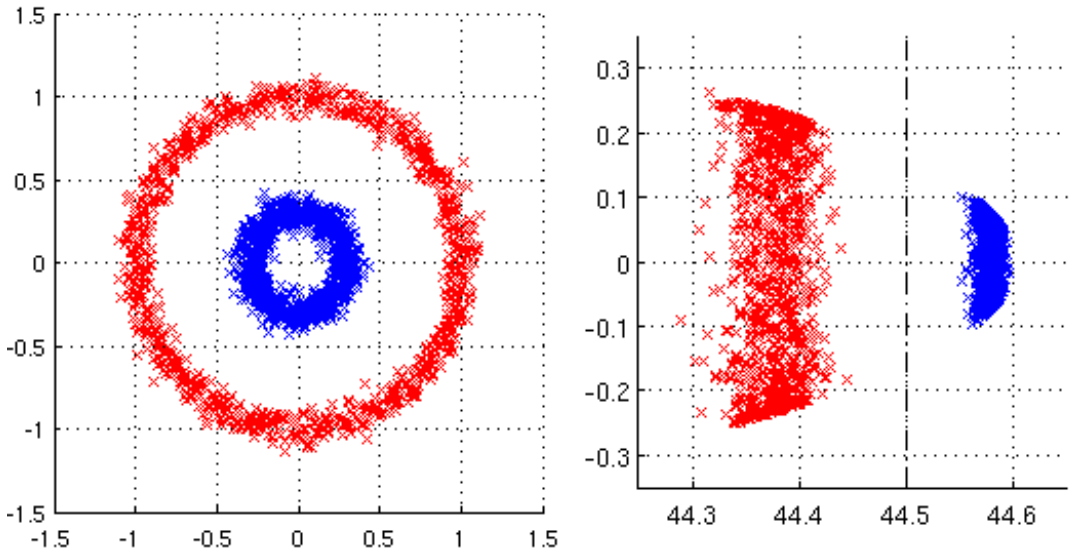
To prove the correctness of the meta-algorithm, see Theorem 4.1 in March et al. [68]. That proof can be adapted from *kd*-trees to general space trees. This representation is a generalization of their algorithms; our meta-algorithm to produces their *kd*-tree and cover tree implementations with a tighter distance bound $B(\mathscr{N}_q)$. Further, the meta-algorithm produces a provably correct dual-tree algorithm with any type of space tree.

7.5 Sparse kernel matrix approximation

The introduction of the kernel trick gave rise to an entire class of kernelized algorithms including kernel principal components analysis (kernel PCA) [111], kernel support vector machines [161], kernel regression [162, 163], spectral clustering [164, 165, 166], Gaussian processes [167, 168], and a variety of other problems. Though each of these methods is significantly different, the commonalities are that a *kernel matrix* K must often be calculated.

Given some positive definite Mercer kernel $\mathcal{K}(\cdot, \cdot)$ and some dataset S_r that contains N points, the kernel matrix $K \in \mathcal{R}^{N \times N}$ is assembled with each element defined as

$$K_{ij} = \mathcal{K}(p_i, p_j) \tag{63}$$



(a) Concentric rings.

(b) After KPCA ($\sigma = 10$).

Figure 33: A standard kernel PCA example.

for p_i and p_j corresponding to points in K. In kernel PCA, for instance, K is then eigen-decomposed and the input points are projected onto some (normally only a few) of the eigenvectors of the kernel matrix.

For kernel PCA, this is superior to regular PCA because nonlinear projections are possible; see Figure 33. This has repeatedly been shown to be an effective technique for a variety of tasks, with applications in handwritten digit recognition [169], image de-noising [170], speech recognition [171, 172], face recognition [173], and a multitude of other machine learning tasks [174, 175, 176].

However, it is important to note that K takes $O(N^2)$ memory and $O(N^2)$ kernel evaluations to compute. In my experience, on modern commodity computing hardware, this means explicit calculation of K is simply impossible for N larger than about 15000.

In this chapter, we examine those situations in which the kernel matrix is sparse or near-sparse, and exploit this fact to develop a dual-tree algorithm which assembles a sparse kernel matrix, for small-bandwidth shift-invariant kernels. As an example, we then apply this to kernel PCA, showing that for small bandwidths, our algorithm outperforms alternatives such as the Nyström method and variants. Although the application of interest here is

kernel PCA[3], the algorithm may be adapted and applied to other kernel methods.

7.5.1 Sparsity in kernel matrices

A sparse (or near-sparse) kernel matrix K can arise if a shift-invariant kernel is used (that is, $\mathcal{K}(x_i, x_j) = \mathcal{K}(\|x_i - x_j\|)$). Genton [177] refers to these kernels as 'stationary kernels'. With a kernel of this type, when two points x_i and x_j are far apart, the kernel evaluation $\mathcal{K}(x_i, x_j)$ approaches zero.

When kernel PCA is viewed as a manifold learning technique, a sparse kernel matrix K is sensible. Similar techniques like IsoMap, LLE, and Laplacian Eigenmaps generate the k-nearest-neighbor graph of the data—which can be transformed to a sparse similarity matrix—in order to represent the manifold the data lie upon. This is intuitive: faraway points are unlikely to lie on the same locally linear manifold region. Thus, interactions between faraway points are not useful, and for kernel PCA a sparse K is reasonable. The connections between kernel PCA and other manifold learning techniques are well-known [178]. Note that Laplacian Eigenmaps corresponds directly to kernel PCA with a sparse kernel matrix.

Of course, a sparse kernel matrix is not always useful. When considering the wider class of all kernel methods (not just kernel PCA), it is known that very small kernel bandwidths can lead to severe loss in performance for some methods [179]. For instance, Murray [180] argues that the small-bandwidth case is mostly irrelevant for Gaussian process regression. On the other hand, it has been shown that there is a definite tradeoff between the sparsity (or near-sparsity) of the kernel matrix and the performance of the method [181]. This relationship can be optimized to provide the best tradeoff between algorithm performance and kernel matrix sparsity.

To show that this is possible for kernel PCA, consider the image de-noising task on the USPS dataset, as in Schölkopf and Smola [111]. In this task we add Gaussian noise to each

[3]Some assumptions must be satisfied for this fast kernel matrix approximation technique to be useful for kernel PCA. A better discussion can be found in Section 7.5.10.

Table 16: Image denoising performance on the USPS dataset as a function of σ.

σ	pSNR	Elements of $K < 0.01$	Elements of $K < 0.001$
1500	13.933 dB	0.00%	0.00%
1000	14.579 dB	0.01%	0.00%
900	14.547 dB	0.88%	0.00%
800	14.421 dB	13.16%	0.04%
700	14.132 dB	56.17%	3.30%
600	13.468 dB	90.75%	38.71%
550	12.712 dB	96.42%	68.55%
500	11.595 dB	98.60%	88.39%
noisy	10.797 dB	–	–

Figure 34: Typical reconstruction; top: clean data, middle: noisy, bottom: after KPCA with $\sigma = 600$.

pixel of each image in the USPS dataset, with variance equal to half the dynamic range of the pixels. Then, we perform kernel PCA, saving 64 of the kernel eigenvectors[4], and reconstruct the images according to Mika et al. [182]. This was done with the Statistical Pattern Recognition Toolbox [183]. For the quality measure, we use the mean pSNR (peak signal-to-noise ratio). Table 16 shows results for various σ values on the full USPS dataset (11k points) using the Gaussian kernel with bandwidth σ. Reconstructions of typical digits are shown in Figure 34.

The tradeoff between pSNR and sparsity of K is clear; but note that we can still obtain reasonable performance with small σ. With $\sigma = 600$, the mean pSNR is only 1 dB lower than peak performance, and K is mostly near-sparse. These results corroborate those shown by Zhang and Genton [181], and also by Mika et al. [182] who showed that de-noising with

[4]64 eigenvectors seemed to provide the best-looking reconstructions; results for other number of eigenvectors exhibited similar trends, though.

kernel PCA is effective when small bandwidths are used.

In addition, if n is increased and points are coming from the same distribution while σ is held constant, then on average, each point will have large kernel interactions with more other points. Thus, it should be possible to reduce σ as n increases.

We can conclude that although performance degrades, a small σ can be chosen to induce a near-sparse K, and at least for kernel PCA, we may still maintain satisfactory performance.

By thresholding small values, though, we are not guaranteed that $\hat{K}$ is positive definite (see Genton [177])[5]. Fortunately, the positive definiteness of $\hat{K}$ is not strictly necessary for eigendecomposition. Further, only the eigenvectors corresponding to large eigenvalues of $\hat{K}$ are interesting for kernel PCA. The eigenvectors of $\hat{K}$ are essentially perturbed eigenvectors of K. So, if the error $K - \hat{K}$ is reasonably small, any eigenvectors of $\hat{K}$ with negative eigenvalues will correspond to eigenvectors of K with small eigenvalues—which would have been ignored anyway. Thus, despite the fact that thresholding can cause $\hat{K}$ to be indefinite, this presents no serious issue, at least for the task of kernel principal components analysis.

7.5.2 Related work on kernel matrix approximation

The problem of large-scale kernel methods has been studied extensively. The most obvious solution is to ignore some of the input points, by generating a smaller kernel matrix on only a cleverly-chosen subset of the data. Smola and Schölkopf [184] suggest three different schemes to select subsets of input data. More recently, Williams and Seeger [24] proposed the Nyström method for subset selection. This approach assumes that K is low-rank, and approximates K as a product of two smaller matrices, significantly reducing storage and computation costs. Another similar approach is the column sampling method [185]. Unfortunately, theoretical work on these algorithms only gives a probabilistic error bound

[5] It would be possible to adapt this algorithm to guarantee that $\hat{K}$ remains positive definite by adapting the techniques of Zhang and Genton [181].

[25, 186]; guaranteed approximation error bounds are not known. Additionally, sampling-based approaches may have problems for datasets with small clusters, where clusters of data with few points may be entirely ignored by the sampling algorithm.

For kernel PCA specifically, another approach is the *Kernel Hebbian Algorithm* [170], an iterative approach to estimate kernel PCA with memory requirements that are linear in the number of points. This type of approach allowed estimation of kernel PCA on the full MNIST training set (60000 points) in 30 to 60 hours using 2007-era computing equipment.

The few approaches that exploit sparsity in the kernel matrix [177, 181] do so only in order to avoid potentially expensive operations that must be performed with the kernel matrix; for instance, in kernel PCA, this involves an $O(N^3)$ eigendecomposition of the kernel matrix. These approaches do not present a solution to the $O(N^2)$ construction of the sparse kernel matrix, and thus still scale poorly with the number of points in the dataset.

More related to this work is that of Gray [83], who applied space partitioning trees to approximate kernels in the context of Gaussian process regression. This approach is related to fast tree-based kernel density estimation [65] and kernel summations [74]. However, Gray's approach does not consider sparsely approximating the kernel matrix and it is not readily adaptable to the kernel PCA problem. In addition, these types of approaches are generally limited to one type of tree (usually *kd*-trees), even though the best tree type is often problem-dependent and dataset-dependent.

7.5.3 A dual-tree algorithm

Like every other algorithm we have discussed in this chapter of the thesis, we will introduce a dual-tree algorithm as a `BaseCase()` and `Score()` function, and it will be tree-independent as a result. The notation, as with all other algorithms, is given in Table 1. The key to the algorithm is determining when we can prune away subtrees of work.

We will use simple thresholding to construct a sparsified kernel matrix approximation $\hat{K}$; that is, given some approximation parameter ϵ, we take $\hat{K}_{ij} = 0$ when $\mathcal{K}(p_i, p_j) \leq \epsilon$. Then, a node combination $(\mathscr{N}_q, \mathscr{N}_r)$ can be pruned when it can be shown that $K_{ij} \leq \epsilon$ for

Algorithm 25 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for sparse kernel matrix approximation.

Input: query node $\mathscr{N}_r$, reference node $\mathscr{N}_r$
Output: a score for the node, or ∞ if the node can be pruned
if $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) \leq \epsilon$ **then**
    return ∞
else if $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) - \mathcal{K}_{\min}(\mathscr{N}_q, \mathscr{N}_r) \leq 2\epsilon$ **then**
    $k_{\text{mid}} \leftarrow \frac{1}{2}\left(\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) - \mathcal{K}_{\min}(\mathscr{N}_q, \mathscr{N}_r)\right)$
    for all $p_q \in \mathscr{D}_q^p$ **and** $p_r \in \mathscr{D}_r^p$ **do**
        $K_{qr} \leftarrow k_{\text{mid}}$
    return ∞
else
    return $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r)$

all x_i that are descendant points of $\mathscr{N}_q$ and all x_j that are descendant points of $\mathscr{N}_r$.

Given two nodes $\mathscr{N}_q$ and $\mathscr{N}_r$ with centers μ_q and μ_r and furthest descendant distances λ_q and λ_r, respectively, we can define the minimum distance between any two descendant points in the two nodes easily:

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) = \|\mu_q - \mu_r\| - \lambda_q - \lambda_r. \tag{64}$$

When the kernel is shift-invariant, it is easy to define the maximum kernel value:

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) = \mathcal{K}\left(d_{\min}(\mathscr{N}_q, \mathscr{N}_r)\right). \tag{65}$$

This allows us to construct a simple `Score()` function, given in Algorithm 25. If the maximum kernel evaluation between any two descendant points is less than or equal to ϵ, then there is no need to recurse into those nodes—all kernel evaluations between descendant points will be approximated as 0.

If a node combination is not pruned, `BaseCase()` is called on combinations of points in the two nodes. Thus, given two points p_q and p_r from an unpruned node combination $(\mathscr{N}_q, \mathscr{N}_r)$, we must set $\hat{K}_{qr}$ to K_{qr}, if $K_{qr} > \epsilon$. A `BaseCase()` function that performs this is given in Algorithm 26.

To run the algorithm, a type of tree and pruning dual-tree traversal are first selected

Algorithm 26 `BaseCase(`p_q`,` p_r`)` for sparse kernel matrix approximation.

1: **Input:** query point p_q, reference point p_r
2: **Output:** none
3: **if** $\mathcal{K}(p_q, p_r) > \epsilon$ **then**
4: $\quad \hat{K}_{qr} \leftarrow \mathcal{K}(p_q, p_r)$

(standard choices might be a *kd*-tree and a dual depth-first traversal). The approximation parameter ϵ is chosen. A tree $\mathscr{T}$ is built on the dataset S and the pruning dual-tree traversal is started with the node combination $(\text{root}(\mathscr{T}), \text{root}(\mathscr{T}))$. The sparse matrix $\hat{K}$ is initialized to all zeros before the traversal.

7.5.4 Correctness proof

Showing the correctness of the algorithm is straightforward and the proof technique resembles correctness proofs for other algorithms.

Theorem 8. *A pruning dual-tree traversal using Algorithm 26 as its* `BaseCase()` *and Algorithm 25 as its* `Score()` *will produce a sparse approximation* $\hat{K}$ *of the true kernel matrix* K *such that* $|K_{ij} - \hat{K}_{ij}| \leq \epsilon$ *for all* $i, j \in \{1, \ldots, n\}$.

Proof. First, assume `Score()` prunes no node combinations. This means `BaseCase()` is called with every possible combination of points $p_q, p_r \in S$. Line 4 of Algorithm 26 means that $K_{qr} = \hat{K}_{qr}$. Otherwise, $|K_{qr} - \hat{K}_{qr}| = |K_{qr}| \leq \epsilon$. So when `Score()` prunes nothing, `BaseCase()` is called on every combination of points, and the result is correct.

Now we show that combinations of points pruned by `Score()` are never outside the ϵ tolerance. By its design, `Score()` only prunes a node combination under two conditions: first, if for every $p_q \in \mathscr{D}_q^p$ and $p_r \in \mathscr{D}_r^p$, it is known that $\mathcal{K}(p_q, p_r) \leq \epsilon$, then the node combination is pruned. In that case, $\hat{K}_{qr} = 0$ and then $|K_{qr} - \hat{K}_{qr}| = |K_{qr}| \leq \epsilon$, which is within tolerance.

Now, consider the second pruning condition: if $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) - \mathcal{K}_{\min}(\mathscr{N}_q, \mathscr{N}_r) \leq 2\epsilon$, then the node combination is pruned, and for every descendant point combination (p_q, p_r), K_{qr} is set to the midpoint of the range, $k_|$ (line 6). Clearly, $\mathcal{K}_{max}(\mathscr{N}_q, \mathscr{N}_r) - k_| \leq \epsilon$, and

$k_| - \mathcal{K}_{\min}(\mathcal{N}_q, \mathcal{N}_r) \leq \epsilon$. Therefore, the approximation for any point is within a tolerance of ϵ, and the approximation satisfies the desired condition. □

Note that Theorem 8 holds for **any** type of space tree and pruning dual-tree traversal.

7.5.5 Application to kernel PCA

In the previous section, we introduced a dual-tree algorithm that can quickly construct a sparse approximation $\hat{K}$ of the kernel matrix K of some dataset S containing n points. A procedure is given below to efficiently perform kernel PCA using this algorithm. ϵ is the user-specified approximation parameter, and $\hat{d}$ is the desired dimension of the nonlinear projections.

1. Construct a tree $\mathcal{T}$ on the dataset S.
2. Initialize $\hat{K}$ to an empty sparse matrix.
3. Run the pruning dual-tree traversal with Algorithm 26 as the `BaseCase()` function and Algorithm 25 as the `Score()` function, and ϵ as the approximation parameter.
4. Use a sparse eigensolver to recover $\hat{d}$ eigenvectors $V = \{v_1, \ldots, v_{\hat{d}}\}$.
5. Construct projected data $\hat{S} = V^T \hat{K}$.

In traditional kernel PCA, the eigendecomposition step (Step 4) takes $O(N^3)$ time[6]. However, because $\hat{K}$ is sparse, a sparse eigensolver implementation such as ARPACK [187] can scale linearly in N, under the assumption that a matrix-vector product can be calculated in $O(N)$ time. This is true when the matrix has $O(N)$ elements. In Section 7.5.6, we show that for certain dataset conditions this is true; then, the eigendecomposition will take $O(N)$ time. In addition, the final projection step (Step 5) takes $O(N^2\hat{d})$ time for traditional kernel PCA, but with a sparse $\hat{K}$, the computations required to compute $V^T\hat{K}$ are far fewer. If $\hat{K}$

[6] A smart eigensolver can do better; in fact, ARPACK can be used with dense matrices, but it will still take $O(N^2)$ time.

has $O(N)$ nonzero elements, then the entire kernel PCA algorithm will take $O(N)$ time (plus the tree construction time, which depends on the type of tree but is often $O(N \log N)$).

7.5.5.1 Centering the kernel matrix

Although K can be approximated effectively and quickly by a sparse matrix $\hat{K}$, often the eigendecomposition is not performed on K but instead a centered version of K [111]:

$$K_c = K - \mathbb{1}_n K - K\mathbb{1}_n + \mathbb{1}_n K \mathbb{1}_n \tag{66}$$

where $\mathbb{1}_n$ is the $n \times n$ matrix filled with $(1/n)$. Fortunately, this poses no issue. The last term, $\mathbb{1}_n K \mathbb{1}_n$, is an $n \times n$ matrix filled entirely with the mean value of K; thus, it can be stored as a single floating-point number. Similarly, $\mathbb{1}_N K$ is a matrix where each row is identical and thus can be stored as a row vector of size n, and $K\mathbb{1}_n$ is a matrix where each column is identical, and can be stored as a column vector of size n.

During eigendecomposition, each Arnoldi iteration requires calculation of $y = \hat{K}x$ for some vector x. $\hat{K}$ is sparse, so this calculation is very fast. If the matrix $\hat{K}$ is centered to produce $\hat{K}_c$, the terms $(\mathbb{1}_n \hat{K})x$, $(\hat{K}\mathbb{1}_n)x$, and $(\mathbb{1}_n \hat{K} \mathbb{1}_n)x$ can all be calculated in $O(n)$ time, preserving the speedup seen when $\hat{K}$ is not centered.

7.5.6 Theoretical results

Like most other dual-tree algorithms, we may use the cover tree and standard cover tree dual-tree traversal to show that, under certain assumptions on the dataset, the entire algorithm will take linear time. These results are in terms of the expansion constant; for more information on the expansion constant and the theoretical properties of the cover tree, see Sections 5.1.1 and 5.2.

We do not need to introduce any new theory to bound the runtime of the sparse kernel matrix approximation method. Instead, runtime bound results for range search and kernel density estimation may be used to produce two different bounds.

First, we must show that sparse kernel matrix approximation may be viewed as an

instance of range search or kernel density estimation. The first pruning rule in Algorithm 25 prunes when

$$\mathcal{K}_{\min}(\mathcal{N}_q, \mathcal{N}_r) \leq \epsilon. \tag{67}$$

The second pruning rule (line 5) can only ever reduce the amount of work performed, so we may ignore that for the purposes of this discussion. If we may assume that $\mathcal{K}(\cdot, \cdot)$ is monotonically decreasing, then we may define $\mathcal{K}^{-1}(\cdot)$ such that

$$\mathcal{K}^{-1}(\alpha) = \|p_i - p_j\| \tag{68}$$

if $\mathcal{K}(p_i, p_j) = \alpha$. Then, our solution (if we ignore the second pruning rule) is equivalent to range search with the range $[0, \alpha]$. As in Section 7.2, if we assume that no column of $\hat{K}$ has more than $|S_{\max}|$ entries and $|S_{\max}|$ is sufficiently small, and also that C (the ratio of the number of points in the α-expansion of $S_{\max}$ to the number of points in $S_{\max}$) is also sufficiently small, then we obtain the following result.

Theorem 9. *Given the above assumptions on the approximated kernel matrix $\hat{K}$, the running time of the sparse kernel matrix approximation dual-tree algorithm using cover trees and the standard cover tree dual-tree traversal is bounded by*

$$O(c_r^{8+\beta}(N + i_t(\mathcal{N}_q) + \theta)), \tag{69}$$

with β defined as in Section 7.2.

Proof. After the assumptions above are applied, this follows directly from Corollary 2. □

Next, we can reduce the algorithm to kernel density estimation, this time ignoring the first pruning rule (line 3). The pruning condition is now

$$\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r) - \mathcal{K}_{\min}(\mathcal{N}_q, \mathcal{N}_r) \leq 2\epsilon \tag{70}$$

and if we assume a shift-invariant, monotonically decreasing kernel this reduces to

$$\mathcal{K}(d_{\min}(\mathcal{N}_q, \mathcal{N}_r)) - \mathcal{K}(d_{\max}(\mathcal{N}_q, \mathcal{N}_r)) \leq 2\epsilon \tag{71}$$

and this is nearly identical to the pruning rule for absolute-value approximate kernel density estimation in Algorithm 21—except the pruning is twice as tight. Now, if we introduce the same assumptions as we did for kernel density estimation, we may adapt the result. In addition to being shift-invariant and monotonically decreasing, it is required that there exists some bandwidth h such that $\mathcal{K}(d)$ must be concave for $d \in [0, h]$ and convex for $d \in [h, \infty)$.

Theorem 10. *Assume that the kernel function $\mathcal{K}(\cdot, \cdot)$ satisfies the assumptions above. Then, given a dataset S_r with expansion constant c_r and using the approximate sparse kernel matrix calculation* `BaseCase()` *and* `Score()` *functions as given in Algorithms 26 and 25, respectively, with the traversal given in Algorithm 8, the running time to calculate the approximate sparse kernel matrix is bounded by*

$$O\left(c_r^{7+\lceil \log_2 \zeta' \rceil}(N + i_t(\mathscr{T}_q) + \theta)\right) \tag{72}$$

with $\zeta' = -\mathcal{K}'(h)\mathcal{K}^{-1}(2\epsilon)\epsilon^{-1}$, $i_t(\mathscr{T}_q)$ defined as in Definition 10, and θ defined as in Lemma 4.

Proof. The discussion before the theorem clarified that the pruning of the `Score()` function given in Algorithm 25 is at least as tight as approximate kernel density estimation with an approximation parameter of 2ϵ. Therefore, under the assumptions of the kernel, we may reuse the results of Theorem 6 and simplify the exponent on the expansion constant c_r:

$$8 + \lceil \log_2 \left(-\mathcal{K}'(h)\mathcal{K}^{-1}(2\epsilon)(2\epsilon)^{-1}\right) \rceil = 7 + \lceil \log_2 \left(-\mathcal{K}'(h)\mathcal{K}^{-1}(2\epsilon)\epsilon^{-1}\right) \rceil \tag{73}$$

and this gives the result. □

Table 17: Datasets used for kernel PCA experiments.

Dataset	Short name	n	d	$\hat{d}$
cloud	cloud	2048	10	2
sat-image	sat	4495	37	3
winequality	wineq	6497	11	3
ISOLET	isolet	7797	617	15
Corel	corel	37749	32	3
MNIST	mnist	70000	784	10
Physics	phy	150000	78	5
Covertype	cover	581012	54	5

The primary difference between this bound and the bound for approximate kernel density estimation is the smaller dependence on c_r; note also that $\mathcal{K}^{-1}(2\epsilon)$ (found in the term ζ') may be much smaller than $\mathcal{K}^{-1}(\epsilon)$ (found in the original term ζ).

7.5.7 Empirical results for kernel PCA

To evaluate the efficiency of our proposed algorithm, we have tested it against other kernel PCA algorithms using shift-invariant kernels. First, to show the algorithm's efficiency with compactly supported kernels, we use the Epanechnikov kernel[7] and perform exact kernel PCA. Then, we demonstrate approximate kernel PCA using the Gaussian kernel. Our algorithms were implemented using **mlpack** [87] in C++ using *kd*-trees with a dual depth-first traversal.

We compare with the MATLAB implementation of an improved Nyström approximation scheme by Zhang et al. [189] and the **mlpack** standard kernel PCA implementation. As suggested by Zhang et al. [189], we use $m = 0.05n$; that is, we use 5% of the points as 'landmark points' for the Nyström method.

Table 17 lists the datasets used in our experiments. They are mostly standard datasets available from the UCI repository [134]. Also listed is the target dimensionality $\hat{d}$ after kernel PCA.

[7]The Epanechnikov kernel is not positive definite, but many finitely supported kernels are conditionally positive definite and thus useful for kernel PCA [188]. It is chosen here not for its performance with kernel PCA but merely as an example of a compactly supported kernel.

Table 18: Results for Epanechnikov kernel.

Data	σ	dt-kpca	density	nyström	kpca
cloud	50	**0.05s**	3.6%	0.09s	29.4s
sat	30	**0.48s**	1.8%	0.59s	247s
wineq	15	**1.29s**	7.1%	1.51s	717s
isolet	10	49.87s	3.3%	**5.03s**	1524s
corel	0.3	**91.0s**	6.1%	267s	fail
mnist	1000	**2417s**	0.9%	fail	fail
phy	5	**238s**	1.1%	fail	fail
cover	225	**239s**	0.1%	fail	fail

Table 18 shows results for the Epanechnikov kernel:

$$\mathcal{K}_e(x, y) = \max\left(0, 1 - \|x - y\|^2/\sigma^2)\right). \tag{74}$$

σ was chosen to be small while still providing reasonable projections for kernel PCA. The runtime of our method ('dt-kpca': dual-tree kernel PCA) is often highly dependent on the sparsity of the kernel matrix, which is a function of σ. Although speedups are lower on high-dimensional datasets (this is typical of tree-based algorithms), it should not be overlooked that our algorithm still outperforms other algorithms for large high-dimensional datasets; other algorithms fail entirely. Performance is poor on the ISOLET and MNIST datasets, likely because those datasets are high-dimensional, and *kd*-trees are known to perform poorly in high dimensions [190]. In either case, when the bandwidth used is such that the kernel matrix is sufficiently sparse, our algorithm can scale to over half a million points without a problem; competing algorithms run out of memory.

A more common kernel choice, though, is the Gaussian kernel:

$$\mathcal{K}_g(x, y) = e^{-\|x-y\|^2/(2\sigma^2)}. \tag{75}$$

The Gaussian kernel has infinite support. Thus, our algorithm will provide an approximate kernel matrix. In this situation, we can compare both runtimes and approximation

Table 19: Results for Gaussian kernel.

Data	σ	dt-kpca	ϵ	density	dt-kpca avg. error	nyström	nyström avg. error	kpca
cloud	25	0.12s	0.001	9.7%	**3.48e-8**	**0.08s**	1.83e-5	29.8s
sat	12	0.72s	0.01	3.5%	**1.69e-7**	**0.395s**	3.57e-6	303.5s
wineq	5	2.10s	0.001	10.6%	**1.25e-8**	**0.92s**	3.41e-6	686.7s
isolet	3	59.04s	0.001	8.1%	**2.05e-8**	**3.804s**	1.43e-6	1550.2s
corel	0.1	**138.4s**	0.005	8.7%	**1.83e-8**	288.8s	2.22e-7	fail
mnist	350	**2842.3s**	0.001	3.0%	**1.84e-8**	fail	n/a	fail
phy	0.75	**330.4s**	0.001	0.9%	n/a	fail	n/a	fail
cover	50	**470.3s**	1e-5	0.1%	n/a	fail	n/a	fail

accuracy between our algorithm and the improved Nyström method. Table 19 shows results. The error measure is $\|\hat{K} - K\|_F/n^2$, is the matrix norm of the difference between the matched eigenvectors; results closer to 0 indicate that the recovered approximate kernel eigenvectors are closer to the true kernel eigenvectors. For larger datasets, standard kernel PCA fails, and computing the errors of either our algorithm or the Nyström method is not possible.

7.5.8 Extensions

7.5.8.1 Any positive definite kernel

Until this point we have only discussed shift-invariant kernels; however, this does not include many popular kernels such as the polynomial kernel ($\mathcal{K}(x, y) = (x^T y)^d$), the linear kernel ($\mathcal{K}(x, y) = x^T y$), or the hyperbolic tangent kernel ($\mathcal{K}(x, y) = \tanh(x^T y)$). It is known that one can bound kernel values for any positive definite kernel by exploiting the triangle inequality in the kernel space [79]:

$$\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r) = \mathcal{K}(\mu_q, \mu_r) + \lambda_q \sqrt{\mathcal{K}(\mu_q, \mu_r)} \tag{76}$$

$$+ \lambda_r \sqrt{\mathcal{K}(\mu_q, \mu_r)} + \lambda_q \lambda_r, \tag{77}$$

$$\mathcal{K}_{\min}(\mathcal{N}_q, \mathcal{N}_r) = \mathcal{K}(\mu_q, \mu_r) - \lambda_q \sqrt{\mathcal{K}(\mu_q, \mu_r)} \tag{78}$$

$$- \lambda_r \sqrt{\mathcal{K}(\mu_q, \mu_r)} - \lambda_q \lambda_r. \tag{79}$$

The use of these bounds requires that $\mathcal{K}(\cdot, \cdot)$ can be evaluated between all μ_q and μ_r in the trees. Thus, each node's centroid must be a point in the dataset. This is not true of *kd*-trees; they cannot be used in this situation. Cover trees [57] provide one alternative, though there are many others, of course; see Chapter 5 and Section 3.6.

The assumption that $\mathcal{K}(\mu_q, \mu_r)$ decreases to 0 as distance between μ_q and μ_r increases is not valid for non-shift-invariant kernels. Thus, the pruning strategy must be to find when $\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r) - \mathcal{K}_{\min}(\mathcal{N}_q, \mathcal{N}_r)$ is small, and then approximate the kernel values. The `Score()` function given in Algorithm 25 can be easily adapted by removing the first pruning condition ($\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r) \leq \epsilon$, in line 3).

However, the main problem with this approach is that there is no guarantee that K is near sparse. Thus, how to avoid the $O(N^2)$ memory requirement for K is unclear, and left as a future challenge.

7.5.9 Application to other kernel methods

The dual-tree algorithm we have proposed is not specific to kernel PCA, although we have proposed it in that context. Most kernel-based algorithms require construction of the kernel matrix K, or at least many kernel evaluations between points in the dataset.

Some examples of these algorithms include kernel discriminant analysis [191], kernel ICA [192], and support vector machines [161]. In settings where K is sparse, or close to sparse, our dual-tree algorithm is easy to adapt and could provide similar speedups.

One challenge here is that the thresholding strategy does not guarantee that the resulting sparse kernel matrix approximation is positive definite; for those kernel methods that require a positive definite kernel matrix, then, the technique of Zhang and Genton [181] for thresholding while maintaining positive definiteness would need to be adapted.

7.5.10 Discussion

This section has described a dual-tree algorithm for quickly approximating a kernel matrix as sparse using thresholding, and has pointed the way toward several extensions and

improvements, and has also demonstrated the effectiveness of the algorithm for the kernel principal components analysis problem. However, it is worth taking a step back to consider those situations in which this algorithm is useful, because it is certainly not useful in all situations; the earlier discussion in Section 7.5.1 touched on this point.

The underlying assumption of the Nyström method is that the kernel matrix is low-rank; that is, it is represented well by $K = GG^T$ where $G \in \mathcal{R}^{N \times r}$ and r, the rank parameter, is far smaller than N. Another way to state the assumption of a low-rank K is to say that there a few large eigenvalues of K, and most are small (or zero); alternately, the eigenspectrum decays quickly. This is also the underlying assumption of kernel PCA: the basis of kernel matrix K can be accurately (though approximately) represented with only a few eigenvectors of K.

But when a kernel matrix K is assembled with a small bandwidth, the matrix becomes block diagonal. Consider the extreme case, where the bandwidth is 0 and the only nonzero elements in K are on the diagonal. If the kernel is shift-invariant then $K = I$, which is certainly not low-rank and cannot be approximated well by a few eigenvectors. Therefore, it is reasonable to say that shrinking the bandwidth of the kernel $\mathcal{K}(\cdot, \cdot)$ will cause the kernel matrix K to become higher-rank and therefore the underlying assumption of kernel PCA is more and more violated.

Are sparsified kernel matrices relevant and useful, then? I would argue yes: as in Section 7.5.1, small-bandwidth kernel machines can still be useful, though often at a performance penalty. Interest has resurfaced recently on small-bandwidth kernels, with the MEKA algorithm [193] and ASKIT [194] garnering recent attention.

Nonetheless, it is important when considering this particular algorithm to know its limitations and assumptions.

7.6 Gaussian mixture model training

In this section, a single-tree algorithm for Gaussian mixture model training is presented. This is a general restatement of the original algorithm by Moore [40], which was limited to *mrkd*-trees; it has been generalized to fit into the tree-independent single-tree algorithm framework, and thus after introducing the problem, we only need to present a `BaseCase()` and `Score()` function.[8]

7.6.1 Problem introduction

The use of Gaussian mixture models is a common machine learning technique to represent complex distributions. Gaussian mixture models are often used in speech recognition applications to represent the distribution of each phoneme. Although a better introduction to GMMs, their uses, and their training is given by both Reynolds [196] and Bilmes [197], we still re-introduce the model briefly and establish our notation. Interested readers who are unfamiliar with GMMs should consult either of those two introductions.

Assume that we are given a dataset $S = \{p_0, p_1, \ldots, p_n\}$, and we wish to fit a Gaussian mixture model with m components to this data. Each component in our Gaussian mixture model θ is described as $c_j = (\phi_j, \mu_j, \Sigma_j)$ for $j \in [0, m)$, where $\phi_j = P(c_j|\theta)$ is the mixture weight of component j, μ_j is the mean of component j, and Σ_j is the covariance of component j. Then, the probability of a point arising from the GMM θ may be calculated as

$$P(p_i|\theta) = \sum_{j=1}^{m} \omega_j (2\pi||\Sigma_j||)^{-1/2} e^{-\frac{1}{2}(p_i-\mu_j)^T \Sigma_j^{-1}(p_i-\mu_j)}. \tag{80}$$

We may also define the probability of a point p_i arising from a particular component in the mixture as

$$a_{ij} := P(p_i|c_j, \theta) = \omega_j (2\pi||\Sigma_j||)^{-1/2} e^{-\frac{1}{2}(p_i-\mu_j)^T \Sigma_j^{-1}(p_i-\mu_j)}. \tag{81}$$

[8] This section is similar to my presentation of this material in a technical report [195].

Then, we may use Bayes' rule to define

$$\omega_{ij} := P(c_j|p_i, \theta) = \frac{a_{ij}\phi_j}{\sum_k a_{ik}\phi_k}. \tag{82}$$

Often, GMMs are trained using an iterative procedure known as the EM (expectation maximization) algorithm, which proceeds in two steps. In the first step, we will compute the probability of each point $p_i \in S$ arising from each component (so, we calculate $a_{ij} = P(p_i|c_j, \theta)$ for all $p_i \in S$ and $c_j \in M$). We can then calculate ω_{ij} using the current parameters of the model θ and the already-calculated a_{ij}. Then, in the second step, we update the parameters of the model θ according to the following rules:

$$\phi_j \leftarrow \frac{1}{n}\sum_{i=0}^{n}\omega_{ij}, \tag{83}$$

$$\mu_j \leftarrow \frac{1}{\sum_{i=0}^{n}\omega_{ij}}\sum_{i=0}^{n}\omega_{ij}p_i, \tag{84}$$

$$\Sigma_j \leftarrow \frac{1}{\sum_{i=0}^{n}\omega_{ij}}\sum_{i=0}^{n}\omega_{ij}(p_i - \mu_j)(p_i - \mu_j)^T. \tag{85}$$

Implemented naively and exactly, we must calculate a_{ij} for each p_i and c_j, giving $O(nm)$ operations per iteration. We can do better with trees, although we will have to introduce some level of approximation.

7.6.2 A generalized single-tree algorithm

We will build a tree, $\mathscr{T}$, on the dataset S, and use this to approximate the values of a_{ij} and ω_{ij} for each i and j. The basic observation is that for any $p_i \in S$, there is likely to be some component (or many components) c_j such that $P(p_i|c_j, \theta)$ (and therefore $P(c_j|p_i, \theta)$) is quite small. Because $P(c_j|p_i, \theta)$ never decays to 0 for finite $\|p_i - \mu_j\|$, we may not avoid any calculations of ω_{ij} if we want to perform the exact EM algorithm.

However, if we allow some amount of approximation, and can determine (for instance) that $\omega_{ij} < \epsilon$, then we can avoid empirically calculating ω_{ij} and simply approximate it as

0. Further, if we can place a bound such that $\zeta - \epsilon < \omega_{ij} < \zeta + \epsilon$, then we can simply approximate ω_{ij} as ζ.

Now, note that for some node $\mathscr{N}_i$, we may calculate $a_j^{\max}$ for some component j, which is an upper bound on the value of a_{ij} for any point $p_i \in \mathscr{D}^p(\mathscr{N}_i)$:

$$a_j^{\max} = (2\pi||\Sigma_j||)^{-1/2} e^{d^M_{\min}(\mathscr{N}_i, \mu_j, \Sigma_j^{-1})} \tag{86}$$

In the equation above, $d^M(\cdot, \cdot, \Sigma^{-1})$ is the Mahalanobis distance:

$$d^M(p_i, p_j, \Sigma^{-1}) = (p_i - p_j)^T \Sigma^{-1} (p_i - p_j) \tag{87}$$

and $d^M_{\min}(\cdot, \cdot, \Sigma^{-1})$ is a generalization of $d_{\min}(\cdot, \cdot)$ to the Mahalanobis distance:

$$d^M_{\min}(\mathscr{N}_i, p_j, \Sigma^{-1}) = \min \left\{ (p_i - p_j)^T \Sigma^{-1} (p_i - p_j) \ , \ p_i \in \mathscr{D}_i^p \right\}. \tag{88}$$

We again assume that we can quickly calculate a lower bound on $d^M_{\min}(\cdot, \cdot, \cdot)$ without checking every descendant point in the tree node. Now, we may use this lower bound to calculate the upper bound $a_j^{\max}$. We may similarly calculate a lower bound $a_j^{\min}$:

$$a_j^{\min} = (2\pi||\Sigma_j||)^{-1/2} e^{d^M_{\max}(\mathscr{N}_i, \mu_j, \Sigma_j^{-1})} \tag{89}$$

with $d^M_{\max}(\cdot, \cdot, \cdot)$ defined similarly to $d^M_{\min}(\cdot, \cdot, \cdot)$. Finally, we can use Bayes' rule to produce the bounds $\omega_j^{\min}$ and $\omega_j^{\max}$ (see Equation 82):

$$\omega_j^{\min} = \frac{a_j^{\min} \phi_j}{a_j^{\min} \phi_j + \sum_{k \neq j} a_k^{\max} \phi_k}, \tag{90}$$

$$\omega_j^{\max} = \frac{a_j^{\max} \phi_j}{a_j^{\max} \phi_j + \sum_{k \neq j} a_k^{\min} \phi_k}. \tag{91}$$

Note that in each of these, we must approximate the term $\sum_k a_{ik} \phi_k$, but we do not know the exact values a_{ik}. Thus, for $\omega_j^{\min}$, we must take the bound $a_{ik} \leq a_k^{\max}$, except for when $j = k$, where we can use the tighter $a_j^{\min}$. Symmetric reasoning applies for the case of $\omega_j^{\max}$.

Algorithm 27 GMM training `BaseCase()`.

Input: model $\theta = \{(\phi_0, \mu_0, \Sigma_0), \ldots, (\phi_{m-1}, \mu_{m-1}, \Sigma_{m-1})\}$, point p_i, partial model $\theta' = \{(\mu'_0, \Sigma'_0), \ldots, (\mu'_{m-1}, \Sigma'_{m-1})\}$, weight estimates $(\omega_0^t, \ldots, \omega_{m-1}^t)$
Output: updated partial model θ'
{Some trees hold points in multiple places; ensure we don't double-count.}
if point p_i already visited **then return**
{Calculate all a_{ij}.}
for all j in $[0, m)$ **do**
    $a_{ij} \leftarrow (2\pi||\Sigma_j||)^{-1/2} e^{-1/2(p_i - \mu_j)^T \Sigma_j^{-1}(p_i - \mu_j)}$
$a_{\text{sum}} \leftarrow \sum_k a_{ik}\phi_k$
{Calculate all ω_{ij} and update model.}
for all j in $[0, m)$ **do**
    $\omega_{ij} \leftarrow \frac{a_{ij}\phi_i}{a_{\text{sum}}}$
    $\omega_j^t \leftarrow \omega_j^t + \omega_{ij}$
    $\mu_j \leftarrow \mu_j + \omega_{ij} p_i$
    $\Sigma_j \leftarrow \Sigma_j + \omega_{ij}(p_i p_i^T)$
return a_{ij}

Now, following the advice of Moore [40], we note that a decent pruning rule is to prune if, for all components j, $\omega_j^{\max} - \omega_j^{\min} < \tau\omega_j^t$, where ω_j^t is a lower bound on the total weight that component j has.

Using that intuition, let us define the `BaseCase()` and `Score()` functions that will define our single-tree algorithm. During our single-tree algorithm, we will have the current model θ and a partial model θ', which will hold unnormalized means and covariances of components. After the single-tree algorithm runs, we can normalize θ' to produce the next model θ.

Algorithm 27 defines the `BaseCase()` function and Algorithm 28 defines the `Score()` function. At the beginning of the traversal, we initialize the weight estimates $\omega_0^t, \ldots, \omega_m^t$ all to 0 and the partial model $\theta' = \{(\mu'_0, \Sigma'_0), \ldots, (\mu'_m, \Sigma'_m)\}$ to 0. At the end of the traversal, we will generate our new model as follows, for each component $j \in [0, m)$:

Algorithm 28 GMM training `Score()`.

Input: model $\theta = \{(\phi_0, \mu_0, \Sigma_0), \ldots, (\phi_{m-1}, \mu_{m-1}, \Sigma_{m-1})\}$, node $\mathcal{N}_i$, weight estimates $(\omega_0^t, \ldots, \omega_{m-1}^t)$, pruning tolerance τ
Output: ∞ if $\mathcal{N}_i$ can be pruned, score for recursion priority otherwise
{Calculate bounds on a_{ij} for each component.}
for all j in $[0, m)$ **do**
    $a_j^{\min} \leftarrow (2\pi||\Sigma_j||)^{-1/2} e^{-1/2(d_{\max}^M(\mathcal{N}_i, \mu_j, \Sigma_j^{-1}))}$
    $a_j^{\max} \leftarrow (2\pi||\Sigma_j||)^{-1/2} e^{-1/2(d_{\min}^M(\mathcal{N}_i, \mu_j, \Sigma_j^{-1}))}$
{Calculate bounds on ω_{ij} for each component.}
for all j in $[0, m)$ **do**
    $\omega_j^{\min} \leftarrow \frac{a_j^{\min}\phi_j}{a_j^{\min}\phi_j + \sum_{k \neq j} a_k^{\max}\phi_k}$
    $\omega_j^{\max} \leftarrow \frac{a_j^{\max}\phi_j}{a_j^{\max}\phi_j + \sum_{k \neq j} a_k^{\min}\phi_k}$
    {Remove parent's prediction for ω_j^t contribution from this node.}
    if $\mathcal{N}_i$ is not the root **then**
        $\omega_j^p \leftarrow$ the value of $\omega_j^{\min}$ calculated by the parent
        $\omega_j^t \leftarrow \omega_j^t - |\mathscr{D}^p(\mathcal{N}_i)|\omega_j^p$
{Determine if we can prune.}
if $\omega_j^{\max} - \omega_j^{\min} < \tau\omega_j^t$ **for all** $j \in [0, m)$ **then**
    {We can prune, so update μ_j and Σ_j.}
    for all j in $[0, m)$ **do**
        $\omega_j^{\text{avg}} \leftarrow 1/2(\omega_j^{\max} + \omega_j^{\min})$
        $\omega_j^t \leftarrow \omega_j^t + |\mathscr{D}^p(\mathcal{N}_i)|\omega_j^{\text{avg}}$
        $c_i \leftarrow$ centroid of $\mathcal{N}_i$
        $\mu_j \leftarrow \mu_j + \omega_j^{\text{avg}} c_i$
        $\Sigma_j \leftarrow \Sigma_j + \omega_j^{\text{avg}} c_i c_i^T$
    return ∞
{Can't prune; update ω_j^t and return.}
for all $j \in [0, m)$ **do**
    $\omega_j^t \leftarrow \omega_j^t + |\mathscr{D}^p(\mathcal{N}_i)|\omega_j^{\min}$
return $1/(\max_{j \in [0,m)} \omega_j^{\max})$

$$\phi_j \leftarrow \frac{1}{n}\omega_j^t \tag{92}$$

$$\mu_j \leftarrow \frac{1}{\omega_j^t}\mu_j' \tag{93}$$

$$\Sigma_j \leftarrow \frac{1}{\omega_j^t}\Sigma_j' \tag{94}$$

After this, the array of ϕ_j values will need to be normalized to sum to 1; this is necessary

because each ω_j^t may be approximate.

To better understand the algorithm, let us first consider the `BaseCase()` function. Given some point p_i, our goal is to update the partial model θ' with the contribution of p_i. Therefore, we first calculate a_{ij} for every component $(\phi_j, \mu_j, \Sigma_j)$. This allows us to then calculate ω_{ij} for each component, and then we may update ω_j^t (our lower bound on the total weight of component j) and our partial model components μ_j' and Σ_j'. Note that in the `BaseCase()` function there is no approximation; if we were to call `BaseCase()` with every point in the dataset, we would end up with μ_j' equal to the result of Equation 84 and Σ_j' equal to the result of Equation 85. In addition, ω_j^t would be an exact lower bound.

Now, let us consider `Score()`, which is where the approximation happens. When we visit a node $\mathscr{N}_i$, our goal is to determine whether or not we can approximate the contribution of all of the descendant points of $\mathscr{N}_i$ at once. As stated earlier, we prune if $\omega_j^{\max} - \omega_j^{\min} < \tau\omega_j^t$ for all components j. Thus, the `Score()` function must calculate $\omega_j^{\max}$ and $\omega_j^{\min}$ (lines 4–10) and make sure ω_j^t is updated.

Keeping ω_j^t correct requires a bit of book-keeping. Remember that ω_j^t is a lower bound on $\sum_i \omega_{ij}$; we maintain this bound by using the lower bound $\omega_j^{\min}$ for each descendant point of a particular node. Therefore, when we visit some node $\mathscr{N}_i$, we must remove the parent's lower bound before adding the lower bound produced with the $\omega_j^{\min}$ value for $\mathscr{N}_i$ (lines 12–14).

Because we have defined our single-tree algorithm as only a `BaseCase()` and `Score()` function, we are left with a generic algorithm. We may use any tree and any traversal (so long as they satisfy the definitions given in Chapter 3).

7.6.3 Possible improvements and extensions

Although we have demonstrated how GMM training can be performed approximately and efficiently with trees, there are still several extensions and improvements that may be performed but are not detailed here:

- A better type of approximation. We are only performing relative approximation using the same heuristic as introduced by Moore [40]. But other types of approximation exist: absolute-value approximation [28], or budgeting [65].

- Provable approximation bounds. In this algorithm, the user selects τ to control the approximation, but there is no derived relationship between τ and the quality of the results. A better user-tunable parameter might be something directly related to the quality of the results; for instance, the user might place a bound on the total mean squared error allowed in μ_j and Σ_j for each j.

- Provable worst-case runtime bounds. Using cover trees, a relationship between the properties of the dataset and the runtime may be derived, similar to other tree-based algorithms which use the cover tree [57, 135].

- Caching during the traversal. During the traversal, quantities such as $a_j^{\min}$, $a_j^{\max}$, $\omega_j^{\min}$, and $\omega_j^{\max}$ for a node $\mathscr{N}_i$ will have some geometric relation to those quantities as calculated by the parent of $\mathscr{N}_i$. These relations could potentially be exploited in order to prune a node without evaluating those quantities. This type of strategy is already in use for nearest neighbor search and max-kernel search in **mlpack**.

Nonetheless, the algorithm, as we have presented it, is generic and flexible and does indeed solve the Gaussian mixture model training problem approximately and efficiently. Our formulation, for *mrkd*-trees, will reduce to Moore's formulation [40], and should perform comparably.

7.7 Max-kernel search

This section introduces the problem of *max-kernel search*, which is a generalized form of similarity search. Until the introduction of this algorithm, there has been no fast, exact algorithm for generalized similarity search (when defined as max-kernel search). Trees can

be used for a fast, exact solution, though; therefore, a single-tree and dual-tree algorithm for fast max-kernel search is developed and introduced.

Although the algorithms in this thesis are generally meant to be tree-independent, unfortunately this particular problem requires some restrictions on the tree. Here, we present an algorithm for cover trees, although it is easily generalizable to similar ball trees, and less easily generalizable to other types of trees (such as cone trees [45]).

This section is a re-presentation of work by myself and Parikshit Ram in SIAM Data Mining 2013 [46] and a later extension of that work [79].

7.7.1 Introduction to max-kernel search

A particularly ubiquitous problem in computer science is that of *max-kernel search*: for a given set S_r of N objects (the *reference set*), a similarity function $\mathcal{K}(\cdot, \cdot)$, and a query object p_q, find the object $p_r \in R$ such that

$$p_r = \underset{p \in S_r}{\operatorname{argmax}} \mathcal{K}(p_q, p). \tag{95}$$

Often, max-kernel search is performed for a large set of query objects S_q.

The most simple approach to this general problem is a linear scan over all the objects in S_r. However, the computational cost of this approach scales linearly with the size of the reference set for a single query, making it computationally prohibitive for large datasets. If $|S_q| = |S_r| = O(N)$, then this approach scales as $O(N^2)$; thus, the approach quickly becomes infeasible for large N.

In our setting we restrict the similarity function $\mathcal{K}(\cdot, \cdot)$ to be a Mercer kernel. As we will see, this is not very restrictive. A Mercer kernel is a positive semidefinite kernel function; these can be expressed as an inner product in some Hilbert space $\mathcal{H}$:

$$\mathcal{K}(x, y) = \langle \varphi(x), \varphi(y) \rangle_{\mathcal{H}}. \tag{96}$$

Often, in practice, the space $\mathcal{H}$ is unknown; thus, the mapping of x to $\mathcal{H}$ ($\varphi(x)$) for any

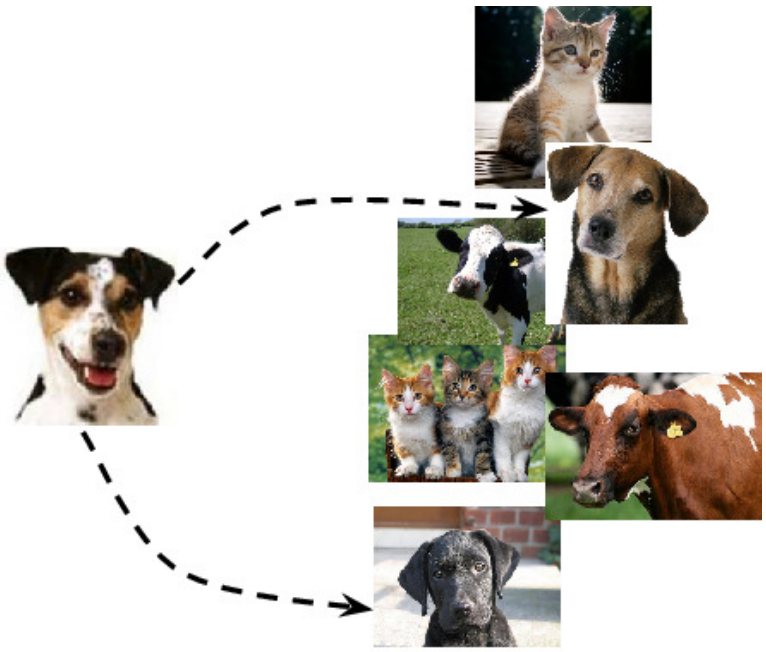


Figure 35: Matching images: an example of max-kernel search.

object x is not known. Fortunately, we do not need to know φ because of the renowned "kernel trick"—the ability to evaluate inner products between any pair of objects in the space $\mathcal{H}$ without requiring the explicit representations of those objects.

Because Mercer kernels do not require explicit representations in $\mathcal{H}$, they are ubiquitous and can be devised for any new class of objects, such as images and documents (which can be considered as points in $\mathcal{R}^d$), to more abstract objects like strings (protein sequences [198], text), graphs (molecules [199], brain neuron activation paths), and time series (music, financial data) [200].

As we mentioned, the max-kernel search problem appears everywhere in computer science and related applications. The widely studied problem of image matching in computer vision is an instance of max-kernel search (Figure 35 presents a simple example). The size of the image sets is continually growing, rendering linear scan computationally prohibitive. Max-kernel search also appears in maximum a posteriori inference [81] as well

as collaborative filtering via the widely successful matrix factorization framework [201]. This matrix factorization obtains an accurate representation of the data in terms of user vectors and item vectors, and the desired result—the user preference of an item—is the inner product between the two respective vectors (this is a Mercer kernel). With ever-scaling item sets and millions of users [202], efficient retrieval of recommendations (which is also max-kernel search) is critical for real-world systems.

Finding similar protein/DNA sequences for a query sequence from a large set of biological sequences is also an instance of max-kernel search with biological sequences as the objects and various domain-specific kernels (for example, the p-spectrum kernel [198], the maximal segment match score [203] and the Smith-Waterman alignment score [204][9]).

An efficient max-kernel search algorithm can be directly applied to biological sequence matching. The field of document retrieval—and information retrieval in general—can be easily seen to be an instance of max-kernel search: for some given similarity function, return the document that is most similar to the query. Spell checking systems are an interesting corollary of information retrieval and also an instance of max-kernel search [205].

A special case of max-kernel search is the problem of nearest neighbor search in metric spaces. In this problem, the closest object to the query with respect to a distance metric is sought. The requirement of a distance metric allows numerous efficient methods for exact and approximate nearest neighbor search, including searches based on space partitioning trees [33, 31, 51, 56, 67, 206] and other types of data structures [150, 156, 132, 157]. However, none of these numerous algorithms are suitable for solving generalized max-kernel search, which is the problem we are considering.

Given the wide applicability of kernels, it is desirable to have a *general* method for efficient max-kernel search that is applicable to any class of objects and corresponding Mercer kernels. To this end, we present a method to accelerate exact max-kernel search for any general class of objects and corresponding Mercer kernels. The specific contributions

[9]These functions provide matching scores for pairs of sequences and can easily be shown to be Mercer kernels.

of this section are listed below.

- The first concept for characterizing the hardness of max-kernel search in terms of the concentration of the kernel values in any direction: the *directional concentration*.
- An single-tree algorithm to index any set of N objects *directly in the Hilbert space defined by the kernel* without requiring explicit representations of the objects in this space.
- Novel single-tree and dual-tree branch-and-bound algorithms on the Hilbert space indexing, which can achieve orders of magnitude speedups over linear search.
- Value-approximate, order-approximate, and rank-approximate extensions to the exact max-kernel search algorithms.
- An $O(N)$ runtime bound for *exact* max-kernel search for $O(N)$ queries with our proposed dual-tree algorithm for *any* Mercer kernel.

7.7.2 Related work

Although there are existing techniques for max-kernel search, almost all of them solve the approximate search problem under restricted settings. The most common assumption is that the objects are in some metric space and the kernel function is *shift-invariant*—a monotonic function of the distance between the two objects ($\mathcal{K}(p, q) = f(\|p - q\|)$). One example is the Gaussian radial basis function (RBF) kernel.

For shift-invariant kernels, a tree-based recursive algorithm has been shown to scale to large datasets for maximum a posteriori inference [81]. However, a shift-invariant kernel is only applicable to objects already represented in some metric space. In fact, max-kernel search with a shift-invariant kernel is equivalent to nearest neighbor search in the input space itself, and can be solved directly using existing methods for nearest neighbor search—an easier and better-studied problem. For shift-invariant kernels, the points can be explicitly embedded in some low-dimensional metric space such that the inner product

between these representations of any two points approximates their corresponding kernel value [207]. This step takes $O(Nd^2)$ time for $S_r \subset \mathcal{R}^d$ and can be followed by nearest neighbor search on these representations to obtain results for max-kernel search in the setting of a shift-invariant kernel.

This technique of obtaining the explicit embedding of objects in some low-dimensional metric space while approximating the kernel function can also be applied to dot-product kernels [208]. Dot-product kernels produce kernel values between any pair of points by operating a monotonic non-decreasing function on their mutual dot-product ($\mathcal{K}(x, y) = f(\langle x, y \rangle)$). Linear and polynomial kernels are simple examples of dot-product kernels. However, this is only applicable to objects which already are represented in some vector space which allows the computation of the dot-products.

Locality-sensitive hashing (LSH) [155] is widely used for image matching, but only with explicitly representable kernel functions that admit a locality sensitive hashing function [209][10]. Kulis and Grauman [210] apply LSH to solve max-kernel search *approximately* for normalized kernels without any explicit representation. Normalized kernels restrict the self-similarity value to a constant ($\mathcal{K}(x, x) = \mathcal{K}(y, y)\ \forall\ x, y \in S$). The preprocessing time for this locality sensitive hashing is $O(p^3)$ and a single query requires $O(p)$ kernel evaluations. Here p controls the accuracy of the algorithm—larger p implies better approximation; the suggested value for p is $O(\sqrt{N})$ with no rigorous approximation guarantees.

A recent work [45] proposed the first technique for exact max-kernel search using a tree-based branch-and-bound algorithm, but is restricted only to linear kernels and the algorithm does not have any runtime guarantees. The authors suggest a method for extending their algorithm to non-representable spaces with general Mercer kernels, but this requires $O(N^2)$ preprocessing time.

There has also been recent interest in similarity search with Bregman divergences

[10]The Gaussian and cosine kernels admit locality sensitive hashing functions with some modifications.

[211], which are non-metrics. Bregman divergences are not directly comparable to kernels, though; they are harder to work with because they are not symmetric like kernels, and are also not as generally applicable to any class of objects as kernel functions. In this paper, we do not address this form of similarity search; Bregman divergences are not Mercer kernels.

7.7.3 Unnormalized kernels

Some kernels used in machine learning are normalized ($\mathcal{K}(x, x) = \mathcal{K}(y, y) \; \forall \; x, y$); examples include the Gaussian and the cosine kernel. As we have discussed, there already exist techniques to solve the max-kernel search problem with normalized kernels.

However, many common kernels like the linear kernel ($\mathcal{K}(x, y) = x^T y$) and the polynomial kernels ($\mathcal{K}(x, y) = (\alpha + x^T y)^d$) for some offset α and degree d) are not normalized. Many applications require unnormalized kernels:

- In the retrieval of recommendations, the normalized linear kernel will result in inaccurate user-item preference scores.
- In biological sequence matching with domain-specific matching functions, $\mathcal{K}(x, x)$ implicitly corresponds to the presence of genetically valuable letters (such as W, H, or P) or not valuable letters (such as X)[11] in the sequence x. This crucial information is lost in kernel normalization.

Therefore, we consider unnormalized kernels. No existing techniques consider unnormalized kernels, and thus no existing techniques can be directly applied to every instance of max-kernel search with general Mercer kernels and any class of objects (Equation 95). Moreover, almost all existing techniques resort to approximate solutions. Not only do our algorithms work for general Mercer kernels instead of just normalized or shift-invariant kernels, but they also provide exact solutions; in addition, extensions to our algorithms for

[11] See the score matrix for letter pairs in protein sequences at `http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt`.

approximation are trivial, and for both the exact and approximate algorithms, we can give asymptotic preprocessing and runtime bounds, as well as rigorous accuracy guarantees for approximate max-kernel searches.

7.7.4 Analysis of the problem

Remember that a Mercer kernel implies that the kernel value for a pair of objects (x, y) corresponds to an inner product between the vector representation of the objects $(\varphi(x), \varphi(y))$ in some Hilbert space $\mathcal{H}$ (see Equation 96). Hence, every Mercer kernel induces the following metric in $\mathcal{H}$:

$$\begin{aligned} d_{\mathcal{K}}(x, y) &= \|\varphi(x) - \varphi(y)\|_{\mathcal{H}} \\ &= \sqrt{\mathcal{K}(x, x) + \mathcal{K}(y, y) - 2\mathcal{K}(x, y)}. \end{aligned} \tag{97}$$

7.7.4.1 Reduction to nearest neighbor search

In situations where max-kernel search can be reduced to nearest neighbor search in $\mathcal{H}$, nearest neighbor search methods for general metrics [157] can be used for efficient max-kernel search. This reduction is possible for normalized kernels. The nearest neighbor for a query p_q in $\mathcal{H}$,

$$\underset{p_r \in S_r}{\operatorname{argmin}}\, d_{\mathcal{K}}(p_q, p_r), \tag{98}$$

is the max-kernel candidate (Equation 95) if $\mathcal{K}(\cdot, \cdot)$ is a normalized kernel. To see this, note that for normalized kernels, $\mathcal{K}(p_q, p_q) = \mathcal{K}(p_r, p_r)$ and thus,

$$d_{\mathcal{K}}(p_q, p_r) = \sqrt{2c - 2\mathcal{K}(p_q, p_r)} \tag{99}$$

where the normalization constant $c = \mathcal{K}(p_q, p_q) = \mathcal{K}(p_r, p_r)$ and is a constant not dependent on p_q or p_r. Therefore, $d_{\mathcal{K}}(p_q, p_r)$ decreases as $\mathcal{K}(p_q, p_r)$ increases, and so $d_{\mathcal{K}}(\cdot, \cdot)$ is minimized when $\mathcal{K}(\cdot, \cdot)$ is maximized. However, the two problems can have very different

answers with unnormalized kernels, because $d_{\mathcal{K}}(p_q, p_r)$ is not guaranteed to decrease as $\mathcal{K}(p_q, p_r)$ increases. As we discussed earlier in Section 7.7.3, unnormalized kernels are an important class of kernels that we wish to consider. Thus, although a reduction to nearest neighbor search is sometimes possible, it is only under the strict condition of a normalized kernel.

7.7.4.2 Hardness of max-kernel search

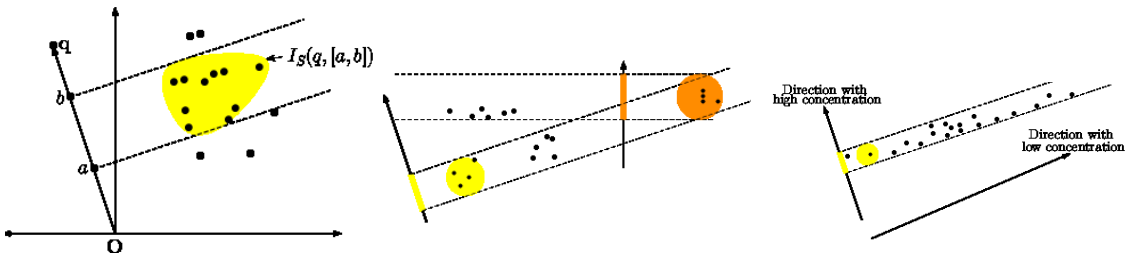
Even if max-kernel search can be reduced to nearest neighbor search, the problem is still hard ($\Omega(N)$ for a single query) without any assumption on the structure of the metric or the dataset S_r. However, better results are possible when assumptions are placed on the expansion constant c of the dataset (see Section 5.2).

The expansion constant effectively bounds the number of points that could be sitting on the surface of a hyper-sphere of any radius around any point. If c is high, nearest neighbor search could be expensive. A value of $c \sim \Omega(N)$ implies that the search cannot be better than linear scan asymptotically. Under the assumption of a bounded expansion constant, though, nearest-neighbor search methods with sublinear or logarithmic theoretical runtime guarantees have been presented [131, 57, 133].

Now, we extend the concept of the expansion concept in order to characterize the difficulty of max-kernel search.

For a given query p_q and Mercer kernel $\mathcal{K}(\cdot, \cdot)$, the kernel values are proportional to the length of the projections in the direction of $\varphi(p_q)$ in $\mathcal{H}$. While the hardness of nearest-neighbor search depends on how concentrated the surface of spheres are (as characterized by the expansion constant), the hardness of max-kernel search should depend on the distribution of the projections in the direction of the query. This distribution can be characterized using the distribution of points in terms of distances:

> *If two points are close in distance, then their projections in any direction are close as well. However, if two points have close projections in a direction, it is not necessary that the points themselves are close to each other.*



(a) Projection interval set. (b) Low value of γ. (c) High value of γ.

Figure 36: Concentration of projections.

It is to characterize this reverse relationship between points (closeness in projections to closeness in distances) that we present a new notion of concentration in any direction:

Definition 13. *Let $\mathcal{K}(x,y) = \langle \varphi(x), \varphi(y) \rangle_{\mathcal{H}}$ be a Mercer kernel, $d_{\mathcal{K}}(x,y)$ be the induced metric from $\mathcal{K}$ (Equation 97), and let $B_S(p,\Delta)$ denote the closed ball of radius Δ around a point p in $\mathcal{H}$. Also, let*

$$I_S(v,[a,b]) = \{r \in S : \langle v, \varphi(r) \rangle_{\mathcal{H}} \in [a,b]\} \tag{100}$$

be the set of objects in S projected onto a direction v in $\mathcal{H}$ lying in the interval $[a,b]$ along v. Then, the **directional concentration constant** *of S with respect to the Mercer kernel $\mathcal{K}(\cdot,\cdot)$ is defined as the smallest $\gamma \geq 1$ such that $\forall u \in \mathcal{H}$ such that $\|u\|_{\mathcal{H}} = 1$, $\forall p \in S$ and $\forall \Delta > 0$, the set*

$$I_S(u, [\langle u, \varphi(p) \rangle_{\mathcal{H}} - \Delta, \langle u, \varphi(p) \rangle_{\mathcal{H}} + \Delta])$$

can be covered by at most γ balls of radius Δ.

The directional concentration constant is not a measure of the number of points projected into a small interval, but rather a measure of the number of "patches" of the data in a particular direction. For a set of points to be close in projections, there are at most γ subsets of points that are close in distances as well. Consider the set of points $B = I_S(q,[a,b])$ projected onto an interval in some direction (Figure 36a). A high value of γ implies that

the number of points in B is high but the points are spread out and the number of balls (with diameter $|b - a|$) required to cover all these points is high as well—with each point possibly requiring an individual ball. Figure 36c provides one such example. A low value of γ implies that either B has a small size or the size of B is high and B can be covered with a small number of balls (of diameter $|b - a|$). Figure 36b is an example of a set with low γ. The directional concentration constant bounds the number of balls of a particular radius required to index points that project onto an interval of size twice the radius.

7.7.5 Indexing points in $\mathcal{H}$

We already know that trees are useful for nearest neighbor search—they have been a primary motivating structure for everything in this thesis—so it should come as no surprise that we may also use trees to perform max-kernel search. However, there are problems we must first overcome.

The first problem, which is the lack of distance metric (remember, $\mathcal{K}(\cdot, \cdot)$ does not satisfy the triangle inequality), is addressed by the induced metric $d_{\mathcal{K}}(\cdot, \cdot)$ in the space $\mathcal{H}$. However, we now have another problem. The standard procedure for constructing *kd*-trees depends on axis-aligned splits along the mean (or median) of a subset of the data in a particular dimension. In $\mathcal{H}$ this does not make sense because we do not have access to the mapping $\varphi(\cdot)$. Thus, *kd*-trees—and any tree that requires knowledge of $\varphi(\cdot)$—cannot be used to index points in $\mathcal{H}$. This includes random projection trees [212] and principal-axis trees [55][12].

Metric trees [213] are a type of space tree that does not require axis-aligned splits. Instead, during construction, metric trees calculate a mean μ for each node [125]. In general, μ is not a point in the dataset the tree is built on. In our situation, we cannot calculate μ because it lies in $\mathcal{H}$ and we do not have access to $\varphi(\cdot)$. However, we can use the kernel trick to avoid calculating μ and evaluate kernels involving μ (assume μ is the mean of node

[12] The explicit embedding techniques mentioned earlier [207, 208] could be used to approximate the mapping $\varphi(\cdot)$ and allow *kd*-trees (and other types of trees) to be used. However, we do not consider that approach in this work.

$\mathcal{N}$, and $\mathcal{D}^p(\mathcal{N})$ refers to the set of descendant points of $\mathcal{N}$):

$$\mathcal{K}(q, \mu) = \frac{\sum_{r \in \mathcal{D}^p(\mathcal{N})} \mathcal{K}(q, r)}{|\mathcal{D}^p(\mathcal{N})|}. \tag{101}$$

This type of trick can also be applied to ball trees and some other similar tree structures. However, it is clear that a single kernel evaluation against the mean is now numerous kernel evaluations, making the use of metric or ball trees computationally prohibitive in our setting, for both tree construction in $\mathcal{H}$ and max-kernel search.

Therefore, we consider the cover tree [57], a recently formulated tree that bears some similarity to the ball tree. The cover tree has been detailed at length throughout this thesis; see Sections 3.6.5 or 5.2 for details.

The cover tree has the interesting property that explicit object representations are unnecessary for tree construction: the tree can be built entirely with only knowledge of the metric function $d_{\mathcal{K}}(\cdot, \cdot)$ evaluated on points in the dataset. Each node $\mathcal{N}_i$ in the cover tree represents a ball in $\mathcal{H}$ with a known radius λ_i and its center μ_i is a point in the dataset. Thus, we can evaluate the minimum distance between two nodes $\mathcal{N}_q$ and $\mathcal{N}_r$ quickly:

$$d_{\min}(\mathcal{N}_q, \mathcal{N}_r) = d_{\mathcal{K}}(\mu_q, \mu_r) - \lambda_q - \lambda_r. \tag{102}$$

Our knowledge of $\mathcal{K}(\cdot, \cdot)$ and its induced metric $d_{\mathcal{K}}(\cdot, \cdot)$ in $\mathcal{H}$, then, is entirely sufficient to construct a cover tree with no computational penalty. In addition to this clear advantage, the time complexities of building and querying a cover tree have been analyzed extensively (see Section 5.2 and [57, 133, 135]), whereas *kd*-trees, metric trees, and other similar structures have been analyzed only under very limited settings [32].

Although we have presented the cover tree as the best tree option, it is not the only option for a choice of tree. What we require of a tree structure is that it can be built only with kernel evaluations between points in the dataset (or distance evaluations)[13]. Therefore,

[13]Earlier, we mentioned kernels that work between abstract objects. For our purposes, it does not make a difference if the kernel works on abstract objects or points, so for simplicity we use the term 'points' instead of 'objects' although the two are essentially interchangeable.

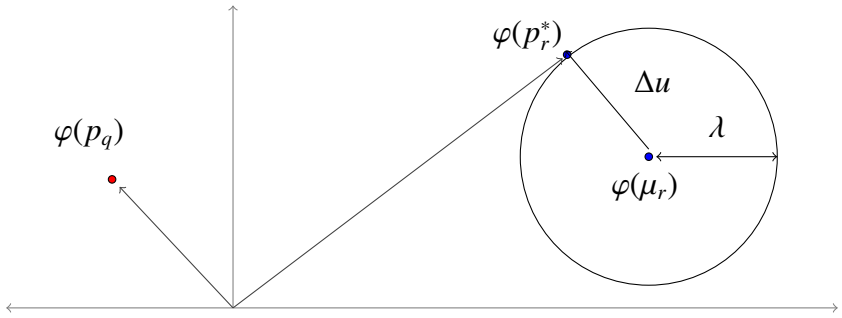


Figure 37: Point-to-node max-kernel upper bound.

we use the notation introduced in Chapter 3 and summarized in Table 1.

An important note is that for cover trees, the center of node $\mathscr{N}_i$, denoted μ_i, is simply the single point held in the cover tree node, p_i (that is, for a node $\mathscr{N}_i$, $\mathscr{P}_i = \{p_i\}$).

7.7.6 Bounding the kernel value

To construct a tree-based algorithm that prunes certain subtrees, we must be able to determine the maximum kernel value possible between a point and any descendant point of a node.

Theorem 11. *Given a space tree node $\mathscr{N}_i$ with center $\varphi(p_i) = \mu_i$ and furthest descendant distance λ_i, the maximum kernel function value between some point p_q and any point in $\mathscr{D}_i^p$ is bounded by the function*

$$\mathcal{K}_{\max}(p_q, \mathscr{N}_i) = \mathcal{K}(p_q, p_i) + \lambda_i \sqrt{\mathcal{K}(p_q, p_q)}. \tag{103}$$

Proof. Suppose that p^* is the best possible match in $\mathscr{D}_i^p$ for p_q, and let $\vec{u}$ be a unit vector in the direction of the line joining $\varphi(p_i)$ to $\varphi(p^*)$ in $\mathcal{H}$. Then,

$$\varphi(p^*) = \varphi(p_i) + \Delta\vec{u} \tag{104}$$

where $\Delta = d_{\mathcal{K}}(\mu_i, p^*)$ is the distance between $\varphi(p_i)$ and the best possible match $\varphi(p^*)$ (see Figure 37). Then, we have the following:

$$\begin{aligned}
\mathcal{K}(p_q, p^*) &= \langle \varphi(p_q), \varphi(p^*) \rangle_{\mathcal{H}} \\
&= \langle \varphi(p_q), \varphi(p_i) + \Delta\vec{u} \rangle_{\mathcal{H}} \\
&= \langle \varphi(p_q), \varphi(p_i) \rangle_{\mathcal{H}} + \langle \varphi(p_q), \Delta\vec{u} \rangle_{\mathcal{H}} \\
&\leq \langle \varphi(p_q), \varphi(p_i) \rangle_{\mathcal{H}} + \Delta \left\| \varphi(p_q) \right\|_{\mathcal{H}}, \qquad (105)
\end{aligned}$$

where the inequality step follows from the Cauchy-Schwartz inequality ($\langle x, y \rangle \leq \|x\| \cdot \|y\|$) and the fact that $\left\|\vec{u}\right\|_{\mathcal{H}} = 1$. From the definition of the kernel function, Equation 105 gives

$$\mathcal{K}(p_q, p^*) \leq \mathcal{K}(p_q, p_i) + \Delta \sqrt{\mathcal{K}(p_q, p_q)}. \qquad (106)$$

We can bound Δ by noting that the distance $d_{\mathcal{K}}(\cdot, \cdot)$ between the center of $\mathcal{N}_i$ and any point in $\mathcal{D}_i^p$ is less than or equal to λ_i. We call our bound $\mathcal{K}_{\max}(p_q, \mathcal{N}_i)$, and the statement of the theorem follows. □

In addition, to construct a dual-tree algorithm, it is useful to extend the maximum point-to-node kernel value of Theorem 11 to the node-to-node setting.

Theorem 12. *Given two space tree nodes $\mathcal{N}_q$ and $\mathcal{N}_r$ with centers $\mu_q = \varphi(p_q)$ and $\mu_r = \varphi(p_r)$, respectively, the maximum kernel function value between any point in $\mathcal{D}_q^p$ and $\mathcal{D}_r^p$ is bounded by the function*

$$\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r) = \mathcal{K}(p_q, p_r) + \lambda_q \sqrt{\mathcal{K}(p_r, p_r)} + \lambda_r \sqrt{\mathcal{K}(p_q, p_q)} + \lambda_q \lambda_r. \qquad (107)$$

Proof. Suppose that $p_q^* \in \mathcal{D}_q^p$ and $p_r^* \in \mathcal{D}_r^p$ are the best possible matches between $\mathcal{N}_q$ and $\mathcal{N}_r$ (see Figure 38 for an illustration); that is,

$$\mathcal{K}(p_q^*, p_r^*) = \max_{p_q \in \mathcal{D}_q^p, p_r \in \mathcal{D}_r^p} \mathcal{K}(p_q, p_r). \qquad (108)$$

Now, let $\vec{u_q}$ be a vector in the direction of the line joining $\varphi(p_q)$ to $\varphi(p_q^*)$ in $\mathcal{H}$, and let $\vec{u_r}$ be a vector in the direction of the line joining $\varphi(p_r)$ to $\varphi(p_r^*)$ in $\mathcal{H}$. Then let $\Delta_q = d_{\mathcal{K}}(p_q, p_q^*)$

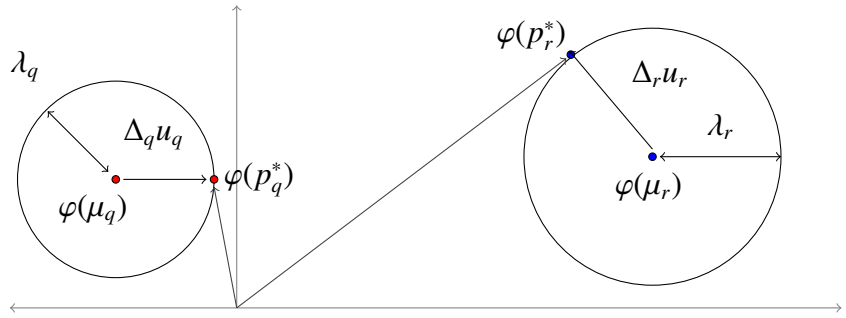


Figure 38: Node-to-node max-kernel upper bound.

and $\Delta_r = d_{\mathcal{K}}(p_r, p_r^*)$. We can use similar reasoning as in the proof for Theorem 11 to show the following:

$$\begin{aligned}
\mathcal{K}(p_q^*, p_r^*) &= \langle \varphi(p_q^*), \varphi(p_r^*) \rangle_{\mathcal{H}} \\
&= \langle \varphi(p_q) + \Delta_q \vec{u_q}, \varphi(p_r) + \Delta_r \vec{u_r} \rangle_{\mathcal{H}} \\
&= \langle \varphi(p_q) + \Delta_q \vec{u_q}, \varphi(p_r) \rangle_{\mathcal{H}} + \langle \varphi(p_q) + \Delta_q \vec{u_q}, \Delta_r \vec{u_r} \rangle_{\mathcal{H}} \\
&= \langle \varphi(p_q), \varphi(p_r) \rangle_{\mathcal{H}} + \langle \Delta_q \vec{u_q}, \varphi(p_r) \rangle_{\mathcal{H}} + \langle \varphi(p_q), \Delta_r \vec{u_r} \rangle_{\mathcal{H}} + \langle \Delta_q \vec{u_q}, \Delta_r \vec{u_r} \rangle_{\mathcal{H}} \\
&\leq \langle \varphi(p_q), \varphi(p_r) \rangle_{\mathcal{H}} + \Delta_q \left\| \varphi(p_r) \right\|_{\mathcal{H}} + \Delta_r \left\| \varphi(p_q) \right\|_{\mathcal{H}} + \Delta_q \Delta_r, \qquad (109)
\end{aligned}$$

where again the inequality steps follow from the Cauchy-Schwarz inequality. We can then substitute in the kernel functions to obtain

$$\mathcal{K}(p_q^*, p_r^*) \leq \mathcal{K}(p_q, p_r) + \Delta_q \sqrt{\mathcal{K}(p_r, p_r)} + \Delta_r \sqrt{\mathcal{K}(p_q, p_q)} + \Delta_q \Delta_r. \qquad (110)$$

Then, as with the point-to-node case, we can bound Δ_q by λ_q and Δ_r can be bounded by λ_r. Call the bound $\mathcal{K}_{\max}(\mathcal{N}_q, \mathcal{N}_r)$, and the statement of the theorem follows. □

For normalized kernels ($\mathcal{K}(x, x) = 1 \ \forall x$)[14], all the points are on the surface of a hypersphere in $\mathcal{H}$. In this case, the above bounds in Theorems 11 and 12 are both correct but possibly loose. Therefore, we can present tighter bounds specifically for this condition.

[14]Earlier we defined normalized kernels as $\mathcal{K}(x, x) = c$ for some constant c, but here for simplicity we consider only $c = 1$. Adapting the proof and bounds for $c \neq 1$ is straightforward.

Theorem 13. *Consider a kernel $\mathcal{K}$ such that $\mathcal{K}(x, x) = 1 \; \forall \; x$, and space tree node $\mathcal{N}_i$ with center $\mu_i = \varphi(p_i)$ and furthest descendant distance λ_i. Define the following quantities:*

$$\alpha_i = \left(1 - \frac{1}{2}\lambda_i^2\right), \tag{111}$$

$$\beta_i = \lambda_i \sqrt{1 - \frac{1}{4}\lambda_i^2}. \tag{112}$$

Then, the maximum kernel function value between some point p_q and any point in $\mathcal{D}_i^p$ is bounded from above by the function

$$\mathcal{K}_{\max}^n(p_q, \mathcal{N}_i) = \begin{cases} \mathcal{K}(p_q, p_i)\alpha_i + \beta_i \sqrt{(1 - \mathcal{K}(p_q, p_i)^2)} & \text{if } \mathcal{K}(p_q, p_i) \leq \alpha_i \\ 1 \text{ otherwise} \end{cases} \tag{113}$$

Proof. Since all the points p_q and $\mathcal{D}_i^p$ are sitting on the surface of a hypersphere in $\mathcal{H}$, $\mathcal{K}(p_q, p)$ denotes the cosine of the angle made by $\varphi(p_q)$ and $\varphi(p)$ at the origin. If we first consider the case where p_q lies within the ball bounding space tree node $\mathcal{N}_i$ (that is, if $d_{\mathcal{K}}(p_q, p_i) < \lambda_i$), it is clear that the maximum possible kernel evaluation should be 1, because there could exist a point in $\mathcal{D}_i^p$ whose angle to p_q is 0. We can restate our condition as a condition on $\mathcal{K}(p_q, p_i)$ instead of $d_{\mathcal{K}}(p_q, p_i)$:

$$\begin{aligned} d_{\mathcal{K}}(p_q, p_i) &< \lambda_i, \\ \sqrt{\mathcal{K}(p_q, p_q) + \mathcal{K}(p_i, p_i) - 2\mathcal{K}(p_q, p_i)} &< \lambda_i, \\ \mathcal{K}(p_q, p_i) &> 1 - \frac{1}{2}\lambda_i^2, \\ \mathcal{K}(p_q, p_i) &> \alpha_i. \end{aligned}$$

Now, for the other case, let $\cos\theta_{p_q p_i} = \mathcal{K}(p_q, p_i)$ and $p^* = \text{argmax}_{p \in \mathcal{D}_i^p} \mathcal{K}(p_q, p)$. Let $\theta_{p_i p^*}$ be the angle between $\varphi(p_i)$ and $\varphi(p^*)$ at the origin, let $\theta_{p_q p^*}$ be the angle between $\varphi(p_q)$ and $\varphi(p^*)$ at the origin, and let $\theta_{p_q p_i}$ be the angle between $\varphi(p_q)$ and $\varphi(p_i)$ at the origin. Then,

$$\begin{aligned}\mathcal{K}(p_q, p^*) &= \cos\theta_{p_q p^*} \\ &\leq \cos(\{\theta_{p_q p_i} - \theta_{p_i p^*}\}_+).\end{aligned}$$

We know that $d_{\mathcal{K}}(p_i, p^*) \leq \lambda_i$, and also that $d_{\mathcal{K}}(p_i, p^*) = \sqrt{2 - 2\cos\theta_{p_i p^*}}$. Therefore, $\cos\theta_{p_i p^*} \geq 1 - \frac{1}{2}\lambda_i^2$. This means

$$\theta_{p_i p^*} \leq |\cos^{-1}(1 - \frac{1}{2}\lambda_i^2)|. \tag{114}$$

Combining this with Equation 114, we get:

$$\mathcal{K}(p_q, p^*) \leq \cos\left([\theta_{p_q p_i} - \theta_{p_i p^*}]_+\right) \tag{115}$$

Now, if we substitute $|\cos^{-1}(1 - \frac{1}{2}\lambda_i^2)|$, the largest possible value for $\theta_{p_i p^*}$, we obtain the following:

$$\mathcal{K}_{\max}(p_q, \mathcal{N}_i) \leq \cos\left(\left[\theta_{q p_i} - \left|\cos^{-1}(1 - \frac{1}{2}\lambda_i^2)\right|\right]_+\right)$$

which can be reduced to the statement of the theorem by the use of trigonometric identities. Combine with the case where $\mathcal{K}(p_q, p_i) > \alpha_i$, and call that bound $\mathcal{K}^n_{\max}(p_q, \mathcal{N}_i)$. Then, the theorem holds. □

We can show a similar tighter bound for the dual-tree case.

Theorem 14. *Consider a kernel $\mathcal{K}$ such that $\mathcal{K}(x, x) = 1\ \forall\ x$, and two space tree nodes $\mathcal{N}_q$ and $\mathcal{N}_r$ with centers $\varphi(p_q) = \mu_q$ and $\varphi(p_r) = \mu_r$, respectively, and furthest descendant distances λ_q and λ_r, respectively. Define the following four quantities:*

$$\alpha_q = \left(1 - \frac{1}{2}\lambda_q^2\right), \tag{116}$$

$$\alpha_r = \left(1 - \frac{1}{2}\lambda_r^2\right), \tag{117}$$

$$\beta_q = \lambda_q \sqrt{1 - \frac{1}{4}\lambda_q^2}, \tag{118}$$

$$\beta_r = \lambda_r \sqrt{1 - \frac{1}{4}\lambda_r^2}. \tag{119}$$

$$\tag{120}$$

Then, the maximum kernel function value between any point in $\mathscr{D}_q^p$ and $\mathscr{D}_r^p$ is bounded from above by the function

$$\mathcal{K}^n_{\max}(\mathscr{N}_q, \mathscr{N}_r) = \begin{cases} \mathcal{K}(p_q, p_r)(\alpha_q\alpha_r - \beta_q\beta_r) + \left(\sqrt{1 - \mathcal{K}(p_q, p_r)^2}\right)\left(\gamma_q\delta_r + \delta_r\gamma_q\right) \\ \quad \textit{if } \mathcal{K}(p_q, p_r) \leq 1 - \frac{1}{2}\left(\lambda_q + \lambda_r\right)^2 \\ 1 \textit{ otherwise}. \end{cases} \tag{121}$$

Proof. All of the points in $\mathscr{D}_q^p$ and $\mathscr{D}_r^p$ are sitting on the surface of a hypersphere in $\mathcal{H}$. This means that $\mathcal{K}(p_q, p_r)$ denotes the cosine of the angle made by $\varphi(p_q)$ and $\varphi(p_r)$ at the origin. Similar to the previous proof, we first consider the case where the balls in $\mathcal{H}$ centered at $\varphi(p_q)$ and $\varphi(p_r)$ with radii λ_q and λ_r, respectively, overlap. This situation happens when $d_{\mathcal{K}}(p_q, p_r) < \lambda_q + \lambda_r$. In this case, it is clear that the maximum possible kernel evaluation should be 1, because there could exist a point in $\mathscr{D}_q^p$ whose angle to a point in $\mathscr{D}_r^p$ is 0. We can restate the condition as a condition on $\mathcal{K}(p_q, p_r)$:

$$\mathcal{K}(p_q, p_r) > 1 - \frac{1}{2}\left(\lambda_q + \lambda_r\right)^2. \tag{122}$$

Now, for the other case, assume that p_q^* and p_r^* are the best matches between points in $\mathscr{D}_q^p$ and $\mathscr{D}_r^p$. Let $\cos\theta_{p_q p_r} = \mathcal{K}(p_q, p_r)$; let $\theta_{p_q p_q^*}$ be the angle between $\varphi(p_q)$ and $\varphi(p_q^*)$ at the origin; similarly, let $\theta_{p_r p_r^*}$ be the angle between $\varphi(p_r)$ and $\varphi(p_r^*)$ at the origin. Lastly, let $\theta_{p_q^* p_r^*}$ be the angle between $\varphi(p_q^*)$ and $\varphi(p_r^*)$ at the origin. Then,

$$\begin{aligned}\mathcal{K}(p_q^*, p_r^*) &= \cos\theta_{p_q^* p_r^*} \\ &\leq \cos\left(\left[\theta_{p_q p_r} - \theta_{p_q p_q^*} - \theta_{p_r p_r^*}\right]_+\right). \end{aligned} \tag{123}$$

Using reasoning similar to the last proof, we obtain the following bounds:

$$\theta_{p_q p_q^*} \leq \left|\cos^{-1}\left(1 - \frac{1}{2}\lambda_q^2\right)\right| \tag{124}$$

$$\theta_{p_r p_r^*} \leq \left|\cos^{-1}\left(1 - \frac{1}{2}\lambda_r^2\right)\right|. \tag{125}$$

We can substitute these two values into Equation 123 to obtain

$$\mathcal{K}(p_q^*, p_r^*) \leq \cos\left(\left[\theta_{p_q p_r} - \left|\cos^{-1}\left(1 - \frac{1}{2}\lambda_q^2\right)\right| - \left|\cos^{-1}\left(1 - \frac{1}{2}\lambda_r^2\right)\right|\right]_+\right). \tag{126}$$

This can be reduced to the statement of the theorem by the use of trigonometric identities. Combine with the conditional from earlier and call the combined bound $\mathcal{K}^n_{\max}(\mathcal{N}_q, \mathcal{N}_r)$. Then, the theorem holds. □

In the upcoming algorithms, we will not use the tighter bounds for normalized kernels given in Theorems 13 and 14; however, it is easy to re-derive the algorithm with the tighter bounds, if a normalized kernel is being used. Simply replace instances of $\mathcal{K}_{\max}(\cdot,\cdot)$ with $\mathcal{K}^n_{\max}(\cdot,\cdot)$.

7.7.7 Single-tree max-kernel search

First, we will present a single-tree algorithm called **single-tree FastMKS** that works on a single query p_q and a reference set S_r. As with all other algorithms in the thesis, we will simply present a `BaseCase()` and `Score()` function, and this is all we need to describe the algorithm (details of this abstraction are the topic of Chapter 3). This allows us to formulate the single-tree algorithm simply and intuitively.

Algorithm 29 `BaseCase(`p_q`,` p_r`)` for FastMKS.

Input: query point p_q, reference point p_r
Output: none

if $\mathcal{K}(p_q, p_r) > k^*$ **then**
    $k^* \leftarrow \mathcal{K}(p_q, p_r)$
    $p^* \leftarrow p_r$

Algorithm 30 `Score(`p_q`,` $\mathscr{N}_r$`)` for FastMKS.

Input: query point p_q, reference space tree node $\mathscr{N}_r$, max-kernel candidate p^* for p_q and corresponding max-kernel value k^*
Output: a score for the node, or ∞ if the node can be pruned

if $\mathcal{K}_{\max}(p_q, \mathscr{N}_r) < k^*$ **then**
    return ∞
else
    return $\mathcal{K}_{\max}(p_q, \mathscr{N}_r)$

In our problem setting, we can prune a node $\mathscr{N}_r$ if no points in $\mathscr{D}_r^p$ can possibly contain a better max-kernel candidate than what has already been found as a max-kernel candidate for p_q. Thus, any descendants of $\mathscr{N}_r$ do not need to be visited, as they cannot improve the solution.

The `BaseCase()` function can be seen in Algorithm 29. It assumes p^* is a global variable representing the current max-kernel candidate and k^* is a global variable representing the current best max-kernel value. The method itself is very simple: calculate $\mathcal{K}(p_q, p_r)$, and if that kernel evaluation is larger than the current best max-kernel value candidate k^*, then store that kernel and p_r as the new best max-kernel candidate and $\mathcal{K}(p_q, p_r)$ as the new best max-kernel value candidate.

The `Score()` function for single-tree FastMKS is given in Algorithm 30. The intuition is clear: if the maximum possible kernel value between p_q and any point in $\mathscr{D}_i^p$ is less than the current max-kernel candidate value, then $\mathscr{N}_i$ cannot possibly hold a better candidate and it can be pruned (return ∞). Otherwise, the kernel value itself is returned. This return value is chosen because pruning single-tree traversals may use the value returned by `Score()` to determine the order in which to visit subsequent nodes [28].

The actual single-tree FastMKS algorithm is constructed by selecting a type of space tree and selecting a pruning single-tree traversal with the `BaseCase()` function as in Algorithm 29 and the `Score()` function as in Algorithm 30. The algorithm is run by building a space tree $\mathscr{T}_r$ on the set of reference points S_r, then using the pruning single-tree traversal with point p_q and tree $\mathscr{T}_q$. At the beginning of the traversal, p^* is initialized to an invalid value and k^* is initialized to $-\infty$.

Proving the correctness of the single-tree FastMKS algorithm is trivial.

Theorem 15. *At the termination of the single-tree FastMKS algorithm for a given space tree and pruning single-tree traversal,*

$$p^* = \underset{p_r \in S_r}{\operatorname{argmax}} \mathcal{K}(p_q, p_r). \tag{127}$$

Proof. First, assume that `Score()` does not prune any nodes during the traversal of the tree $\mathscr{T}_r$. Then, by the definition of pruning single-tree traversal, `BaseCase()` is called with p_q and every $p_r \in S_r$. This is equivalent to linear scan and will give the correct result.

Then, by Theorem 11 (or Theorem 13 if $\mathcal{K}(\cdot, \cdot)$ is normalized and $\mathcal{K}^n_{\max}(\cdot, \cdot)$ is being used), a node is only pruned if it does *not* contain a point p_r where $\mathcal{K}(p_q, p_r) > k^*$. Thus, `BaseCase()` is only *not* called in situations where p^* and k^* would not be modified. This, combined with the previous observation, means that p^* and k^* are equivalent to the linear scan results at the end of the traversal—and we know the linear scan results are correct. Thus, the theorem holds. □

In the original publications, an $O(\log N)$ per query runtime bound was claimed. However, this proof depends on the original nearest neighbor runtime proof for cover trees, which I now believe to be either unclear or incorrect (see Section 5.3). Therefore, I have omitted this runtime bound.

7.7.8 Dual-tree fast max-kernel search

Now, we present a dual-tree algorithm for max-kernel search, called **dual-tree FastMKS**. This algorithm, as with the single-tree algorithm in Section 7.7.7, is presented as only a `BaseCase()` and `Score()` function; in this case, `BaseCase()` remains the same (Algorithm 29).

Because the dual-tree algorithm solves max-kernel search for an entire set of query points S_q, we must store a kernel candidate p^* and value k^* for each query point p_q; call these $p^*(p_q)$ and $k^*(p_q)$, respectively. At the initialization of the algorithm $k^*(p_q) = \infty$ for each $p_q \in S_q$ and $p^*(p_q)$ is set to some invalid point.

The pruning rule is slightly more complex. In the dual-tree setting, we can only prune a node combination $(\mathscr{N}_q, \mathscr{N}_r)$ if and only if $\mathscr{D}_r^p$ contains *no* points that can improve $p^*(p_q)$ and $k^*(p_q)$ for any $p_q \in \mathscr{D}_q^p$. There are multiple ways to express this concept, and we will use two of them to construct a bound function to determine when we can prune. This section is heavily based on the reasoning used to derive the nearest-neighbor search bound; see Section 7.1 and [28].

First, consider the smallest max-kernel value $k^*(p_q)$ for all points $p_q \in \mathscr{D}_q^p$; call this $B_1(\mathscr{N}_q)$:

$$\begin{aligned} B_1(\mathscr{N}_q) &= \min_{p_q \in \mathscr{D}_q^p} k^*(p_q) \\ &= \min \left\{ \min_{p_q \in \mathscr{P}_q} k^*(p_q), \min_{\mathscr{N}_c \in \mathscr{C}_q} B_1(\mathscr{N}_q) \right\} \end{aligned}$$

where the simplification is a result of expressing $B_1(\mathscr{N}_q)$ recursively. Now, note also that for any point $p_q \in \mathscr{D}_q^p$ with max-kernel candidate value $k^*(p_q)$, we can place a lower bound on the true max-kernel value $\hat{k}(p'_q)$ for any $p'_q \in \mathscr{D}_r^p$ by bounding $\mathcal{K}(p'_q, p^*(p_q))$. This gives

$$\hat{k}(p'_q) \geq k^*(p_q) - (\rho_q + \lambda_q) \sqrt{\mathcal{K}\left(p^*(p_q), p^*(p_q)\right)}$$

where ρ_q is the maximum distance from any $p \in \mathscr{P}_q$ to the centroid of $\mathscr{N}_q$ (for cover trees,

Algorithm 31 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for FastMKS.

1: **Input:** query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
2: **Output:** a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ or ∞ if the combination can be pruned
3: **if** $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q)$ **then**
4: **return** ∞
5: **else**
6: **return** $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r)$

this value is always 0). This inequality follows using similar reasoning as Theorem 11, except for that we are finding a lower bound instead of an upper bound.

Considering all the points $p_q \in \mathscr{D}_q^p$, we find that the minimum possible max-kernel value for any point p_q can be expressed as

$$\max_{p_q \in \mathscr{D}_q^p} k^*(p_q) - (\rho_q + \lambda_q) \sqrt{\mathcal{K}\left(p^*(p_q), p^*(p_q)\right)}.$$

However, this is difficult to calculate in practice; thus, we introduce a second bounding function that can be quickly calculated by only considering points in $\mathscr{P}_q$ and not $\mathscr{D}_q^p$:

$$B_2(\mathscr{N}_q) = \max_{p_q \in \mathscr{P}_q} k^*(p_q) - (\rho_q + \lambda_q) \sqrt{\mathcal{K}\left(p^*(p_q), p^*(p_q)\right)}.$$

Now, we can take the better of $B_1(\mathscr{N}_q)$ and $B_2(\mathscr{N}_q)$ as our pruning bound:

$$B(\mathscr{N}_q) = \max\left\{B_1(\mathscr{N}_q), B_2(\mathscr{N}_q)\right\}. \tag{128}$$

This means that we can prune a node combination $(\mathscr{N}_q, \mathscr{N}_r)$ if

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q),$$

and therefore we introduce a `Score()` function in Algorithm 31 that uses $B(\mathscr{N}_q)$ to determine if a node combination should be pruned.

As with the single-tree algorithm, a correctness proof is straightforward.

Theorem 16. *At the termination of the dual-tree FastMKS algorithm for a given space tree and pruning dual-tree traversal,*

$$p^*(p_q) = \operatorname*{argmax}_{p_r \in S_r} \mathcal{K}(p_q, p_r) \; \forall \; p_q \in S_q. \tag{129}$$

Proof. First, assume that `Score()` does not prune any node combinations during the dual traversal of the trees $\mathscr{T}_q$ and $\mathscr{T}_r$. Then, by the definition of pruning dual-tree traversal, `BaseCase()` will be called with each $p_q \in S_q$ and each $p_r \in S_r$; this is equivalent to linear scan and will give the correct results.

We have already stated the validity of $B(\mathscr{N}_q)$ (Equation 128). Because of that, and also by Theorem 12 (or Theorem 14 if $\mathcal{K}(\cdot, \cdot)$ is normalized and $\mathcal{K}^n_{\max}(\cdot, \cdot)$ is being used), a node combination is only pruned is it does *not* contain a point p_r that would modify $p^*(p_q)$ or $k^*(p_q)$ for any $p_q \in \mathscr{D}^p_q$. This, combined with the previous observation, means that p^* and k^* are equivalent to the linear scan results for each $p_q \in S_r$, and thus, the theorem holds. □

7.7.9 Dual-tree algorithm runtime analysis

Now, we bound the runtime of dual-tree FastMKS using the same adaptive algorithm analysis techniques for the cover tree as in other sections. The original formulation of this bound depended on a virtually ununderstandable quantity called the *inverse constant of bichromaticity* [79], which is related to the similarly ununderstandable *constant of bichromaticity* from a few years prior [133]. Here, we re-derive the FastMKS runtime bound using Theorem 1, from Section 5.2 (or [135]), which provides a much easier to understand bound that depends on the imbalance of the tree and a few other quantities.

As with all other adaptive analysis runtime bounds in this thesis, we restrict our consideration of the dual-tree algorithm to the cover tree and the standard cover tree dual-tree traversal (Algorithm 8).

We still must introduce a handful of additional quantities, though. First, we define the maximum norms and minimum norms of the query set S_q and reference set S_r:

$$\eta_q = \max_{p_q \in S_q} \|\varphi(p_q)\|_{\mathcal{H}}, \tag{130}$$

$$\eta_r = \max_{p_r \in S_r} \|\varphi(p_r)\|_{\mathcal{H}}, \tag{131}$$

$$\tau_q = \min_{p_q \in S_q} \|\varphi(p_q)\|_{\mathcal{H}}, \tag{132}$$

$$\tau_r = \min_{p_r \in S_r} \|\varphi(p_r)\|_{\mathcal{H}}. \tag{133}$$

Next, we use these quantities to place bounds on the maximum distances $d_{\mathcal{H}}(\cdot, \cdot)$ between points in the dataset, and place an upper bound on the maximum scale of cover tree nodes.

Lemma 5. *For the query set S_q, the maximum distance between any points in S_q,*

$$d_{\mathcal{H}}^{\max}(S_q) \leq 2\eta_q. \tag{134}$$

Proof. We can alternately write $d_{\mathcal{H}}^{\max}(S_q)$ as

$$\begin{aligned} d_{\mathcal{H}}^{\max}(S_q) &= \max_{p_i \in S_q, p_j \in S_q} d_{\mathcal{H}}(p_i, p_j) \\ (d_{\mathcal{H}}^{\max}(S_q))^2 &= \max_{p_i \in S_q, p_j \in S_q} \|\varphi(p_i)\|_{\mathcal{H}}^2 + \|\varphi(p_j)\|_{\mathcal{H}}^2 - 2\langle \varphi(p_i), \varphi(p_j) \rangle_{\mathcal{H}}. \end{aligned}$$

Note that $\langle \varphi(p_i), \varphi(p_j) \rangle_{\mathcal{H}}$ is minimized when $\varphi(p_i)$ and $\varphi(p_j)$ point opposite ways in $\mathcal{H}$: $\varphi(p_i)/\|\varphi(p_i)\|_{\mathcal{H}} = -(\varphi(p_j)/\|\varphi(p_j)\|_{\mathcal{H}})$. Thus,

$$\begin{aligned} (d_{\mathcal{H}}^{\max}(S_q))^2 &\leq \max_{p_i \in S_q, p_j \in S_q} \|\varphi(p_i)\|_{\mathcal{H}}^2 + \|\varphi(p_j)\|_{\mathcal{H}}^2 \\ &\qquad - 2\max\{\langle \varphi(p_i), -\varphi(p_i) \rangle_{\mathcal{H}}, \langle \varphi(p_j), -\varphi(p_j) \rangle_{\mathcal{H}}\} && (135) \\ &\leq \max_{p_i \in S_q, p_j \in S_q} \|\varphi(p_i)\|_{\mathcal{H}}^2 + \|\varphi(p_j)\|_{\mathcal{H}}^2 + 2\max\{\|\varphi(p_i)\|_{\mathcal{H}}^2, \|\varphi(p_j)\|_{\mathcal{H}}^2\} && (136) \\ &\leq 4\eta_q^2. && (137) \end{aligned}$$

This trivially reduces to the result. □

Corollary 3. *The maximum distance between any points in S_r is*

$$d_{\mathcal{H}}^{\max}(S_r) \leq 2\eta_r. \tag{138}$$

Lemma 6. *The top scale s_r^T (maximum/largest scale) in the cover tree $\mathscr{T}_r$ built on S_r is bounded as*

$$s_r^T \leq \log_2(\eta_r). \tag{139}$$

Proof. The root of the tree $\mathscr{T}_r$ is the node with the largest scale, and it is the only node of that scale (call this scale s_r^T). The furthest descendant distance of the root node is bounded by $2^{s_r^T+1}$; however, this is not necessarily the distance between the two furthest points in the dataset (consider a tree where the root node is near the centroid of the data). This, with Corollary 3, yields $2^{s_r^T+1} \leq 2\eta_r$ which is trivially reduced to the result. □

Finally, we are ready to show the main result of the section.

Theorem 17. *Given a Mercer kernel $\mathcal{K}(\cdot,\cdot)$, a reference set S_r of size N with expansion constant c_r and directional concentration constant γ_r, a query set S_q of size $O(N)$, and with α defined as*

$$\alpha = 1 + \frac{2\eta_r}{\tau_q}, \tag{140}$$

the dual-tree FastMKS algorithm using cover trees and the standard dual-tree cover tree traversal on $\mathscr{T}_q$ (a cover tree built on S_q) and $\mathscr{T}_r$ (a cover tree built on S_r) with $i_t(\cdot)$ defined as in Definition 10 and θ defined as in Lemma 4 requires time

$$O\left(\gamma_r c_r^{(7\log_2\alpha)}(N + i_t(\mathscr{T}_q) + \theta)\right) \tag{141}$$

Proof. We know from Theorem 1 that the running time of any dual-tree algorithm which uses the cover tree and the standard cover tree dual-tree traversal is $O(c_r^4|R^*|\chi\psi(N + i_t(\mathscr{T}_q) + \theta))$. Our only job, then, is to fill in each of these quantities and simplify.

Smart caching strategies allow `BaseCase()` and `Score()` to be written such that they each take $O(1)$ time, meaning that $\chi = \psi = O(1)$ and we have no further need to consider those terms. For `Score()`, the primary calculation is that of $B(\mathscr{N}_q)$. This can be made $O(1)$ by caching the values of both $B_1(\mathscr{N}_q)$ and $B_2(\mathscr{N}_q)$ and only updating when necessary: changes to $B_2(\mathscr{N}_q)$ for a node are propagated upwards, and changes to $B_1(\mathscr{N}_q)$ only require an $O(1)$ check anyway because each cover tree node has only one point.

The last thing, then, is to bound $|R^*|$, the size of the largest reference set; this turns out to be quite in-depth. Consider some reference set R encountered with maximum reference scale $s_r^{\max}$ and query node $\mathscr{N}_q$. Every node $\mathscr{N}_r \in R$ satisfies the property enforced in line 10 that

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) \geq B(\mathscr{N}_q). \tag{142}$$

Remembering that $\sqrt{\mathcal{K}(p, p)} = \|\varphi(p)\|_{\mathcal{H}}$, we can relax $B(\mathscr{N}_q)$ (Equation 128) for the cover tree (where $\rho_i = 0$ for all $\mathscr{N}_i$) to show

$$\begin{aligned} B(\mathscr{N}_q) &\geq \max_{p \in \mathscr{P}_q} \left(k^*(p) + \lambda_q \|\varphi(p^*(p))\|_{\mathcal{H}} \right) \\ &= k^*(p_q) - \lambda_q \left\| \varphi(p^*(p_q)) \right\|_{\mathcal{H}} \end{aligned} \tag{143}$$

which we can combine with Equation 142 to obtain

$$\begin{aligned} \mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) &\geq k^*(p_q) + \lambda_q \|\varphi(p_q)\|_{\mathcal{H}} \\ \mathcal{K}(p_q, p_r) &\geq k^*(p_q) - \lambda_q(\|\varphi(p_r)\|_{\mathcal{H}} + \|\varphi(p^*(p_q))\|_{\mathcal{H}}) - \lambda_r \|\varphi(p_q)\|_{\mathcal{H}} - \lambda_q \lambda_r \end{aligned} \tag{144}$$

and, remembering that the scale of $\mathscr{N}_q$ is s_q and the scale of $\mathscr{N}_r$ is bounded above by $s_r^{\max}$, we simplify further to

$$\mathcal{K}(p_q, p_r) \geq k^*(p_q) - 2^{s_q+1}(\|\varphi(p_r)\|_{\mathcal{H}} + \|\varphi(p^*(p_q))\|_{\mathcal{H}}) - 2^{s_r^{\max}+1} \|\varphi(p_q)\|_{\mathcal{H}} - 2^{s_q + s_r^{\max}+2}. \tag{145}$$

We can express this conditional as membership in a set I_{S_r} by first defining the true maximum kernel value for p_q as

$$\hat{k}(p_q) = \max_{p_r \in S_r} \mathcal{K}(p_q, p_r). \tag{146}$$

The condition (Equation 145) can be stated as membership in a set:

$$\varphi(p_r) \in I_{S_r}\Big(\varphi(p_q), [b_l, \hat{k}(p_q)]\Big) \tag{147}$$

where

$$b_l = k^*(p_q) - 2^{s_q+1}(\|\varphi(p_r)\|_{\mathcal{H}} + \|\varphi(p^*(p_q))\|_{\mathcal{H}}) - 2^{s_r^{\max}+1}\|\varphi(p_q)\|_{\mathcal{H}} - 2^{s_q+s_r^{\max}+2}. \tag{148}$$

Now, we produce a lower bound for b_l. Note that $\hat{k}(p_q) \leq k^*(p_q) + 2^{s_r^{\max}+1}\|\varphi(p_q)\|_{\mathcal{H}}$, and see

$$\begin{aligned} b_l &\geq \hat{k}(p_q) - 2^{s_q+1}(\|\varphi(p_r)\|_{\mathcal{H}} + \|\varphi(p^*(p_q))\|_{\mathcal{H}}) - 2^{s_r^{\max}+2}\|\varphi(p_q)\|_{\mathcal{H}} - 2^{s_q+s_r^{\max}+2} && (149)\\ &\geq \hat{k}(p_q) - 2^{s_r^{\max}+1}(\|\varphi(p_r)\|_{\mathcal{H}} + \|\varphi(p^*(p_q))\|_{\mathcal{H}}) - 2^{s_r^{\max}+2}\|\varphi(p_q)\|_{\mathcal{H}} - 2^{2s_r^{\max}+2} && (150)\end{aligned}$$

which follows because $s_q < s_r^{\max}$ during a reference recursion (see line 4). Using the maximum and minimum norms defined earlier, we can bound b_l further:

$$\begin{aligned} b_l &\geq \hat{k}(p_q) - 2^{s_r^{\max}+1}(\eta_r + \eta_r) - 2^{s_r^{\max}+2}\|\varphi(p_q)\|_{\mathcal{H}} - 2^{2s_r^{\max}+2} && (151)\\ &= \hat{k}(p_q) - 2^{s_r^{\max}+2}(\|\varphi(p_q)\|_{\mathcal{H}} + \eta_r + 2^{s_r^{\max}}) && (152)\\ &\geq \mathcal{K}(p_q, p_r) - 2^{s_r^{\max}+2}(\|\varphi(p_q)\|_{\mathcal{H}} + \eta_r + 2^{s_r^{\max}}) && (153)\\ &\geq \mathcal{K}(p_q, p_r) - 2^{s_r^{\max}+2}(\|\varphi(p_q)\|_{\mathcal{H}} + \eta_r + 2^{s_r^{T}}) && (154)\\ &\geq \mathcal{K}(p_q, p_r) - 2^{s_r^{\max}+2}(\|\varphi(p_q)\|_{\mathcal{H}} + 2\eta_r) && (155)\end{aligned}$$

where the last two bounding steps result from Lemma 6.

Now, note that

$$\frac{b_l}{\|\varphi(p_q)\|_{\mathcal{H}}} \geq \langle u, \varphi(p_r)\rangle_{\mathcal{H}} - 2^{s_r^{\max}+2}\left(1 + \frac{\eta_r + 2^{s_r^T}}{\tau_q}\right) \tag{156}$$

then set $\alpha = 1 + (2\eta_r/\tau_q)$ (α is not dependent on the scale $s_r^{\max}$; this is important) and use the conditional from Equation 147 to get

$$\begin{aligned}
\varphi(p_r) &\in I_{S_r}(\varphi(p_q), [b_l, \hat{k}(p_q)]) && (157)\\
&\subseteq I_{S_r}(\varphi(p_q), [b_l, \mathcal{K}(p_q, p_r) + 2^{s_r^{\max}+1}\|\varphi(p_q)\|_{\mathcal{H}}]) && (158)\\
&\subseteq I_{S_r}(u, [\langle u, \varphi(p_r)\rangle_{\mathcal{H}} - 2^{s_r^{\max}+2}\alpha, \langle u, \varphi(p_r)\rangle_{\mathcal{H}} + 2^{s_r^{\max}+1}]) && (159)\\
&\subseteq I_{S_r}(u, [\langle u, \varphi(p_r)\rangle_{\mathcal{H}} - 2^{s_r^{\max}+2}\alpha, \langle u, \varphi(p_r)\rangle_{\mathcal{H}} + 2^{s_r^{\max}+2}\alpha]). && (160)
\end{aligned}$$

This is true for each point p_i of each node $\mathcal{N}_i$ in R_i. Thus, if we can place a bound on the number of points in the set given in Equation 160, then we are placing a bound on $|R_i|$ for any scale s_i. To this end, we can use the definition of directional concentration constant, to show that there exist γ_r points $p_j \in S_r$ such that

$$I_{S_r}(u, [\langle u, \varphi(p_r)\rangle_{\mathcal{H}} - 2^{s_r+2}\alpha, \langle u, \varphi\rangle(p_r)_{\mathcal{H}} + 2^{s_r+2}\alpha]) \subseteq \bigcup_{j=1}^{\gamma_r} B_{S_r}(p_j, 2^{s_r+2}\alpha). \tag{161}$$

By Lemma 2, each point p_r of each node $\mathcal{N}_r \in R$ must be separated by at least $2^{s_r^{\max}}$, because each point in R must have a parent with scale at least $s_r^{\max}+1$. Thus, we must bound the number of balls of radius $2^{s_r^{\max}-1}$ that can be packed into the set defined by Equation 161. For each p_j, we have

$$\begin{aligned}
|B_{S_r}(p_j, 2^{s_r^{\max}+2}\alpha)| &\leq c_r^2 |B_{S_r}(p_j, 2^{s_r^{\max}-1}\alpha)|\\
&\leq c_r^{3\log_2\alpha} |B_{S_r}(p_j, 2^{s_r^{\max}-1})|.
\end{aligned}$$

This allows us to conclude that $|R^*| \leq \gamma_r c_r^{(3\log_2\alpha)}$ and therefore the total running time of the algorithm is $O(\gamma_r c_r^{(7\log_2\alpha)} \nu N)$, and the theorem holds. □

Note that if dual-tree FastMKS is being run with the same set as the query set and reference set (i.e. monochromatic search), $\theta = 0$, yielding a tighter bound.

7.7.10 Extensions for approximate max-kernel search

For further scalability, we can develop an extension of FastMKS that does not return the exact max-kernel value but instead an approximation thereof. Even though we are focusing on exact max-kernel search, we wish to demonstrate that the tree based method can be very easily extended to perform the approximate max-kernel search. For any query p_q, we are seeking $\hat{p}(p_q) = \arg\max_{p_r \in S_r} \mathcal{K}(p_q, p_r)$. Let $\mathcal{K}(p_q, \hat{p}(p_q)) = \hat{k}(p_q)$ (as before). Then approximation can be achieved in the following ways:

1. Absolute value approximation: for all queries $p_q \in S_q$, find $p_r \in S_r$ such that $\mathcal{K}(p_q, p_r) \geq \hat{k}(p_q) - \epsilon$ for some $\epsilon > 0$.

2. Relative value approximation: for all queries $p_q \in S_q$, find $p_r \in S_r$ such that $\mathcal{K}(p_q, p_r) \geq (1 - \epsilon)\hat{k}(p_q)$ for some $\epsilon > 0$[15].

3. Rank approximation: return $p_r \in S_r$ such that $\left|\{p'_r \in S_r : \mathcal{K}(p_q, p'_r) > \mathcal{K}(p_q, p_r)\}\right| \leq \tau$.

The following three subsubsections present how both single-tree FastMKS and dual-tree FastMKS can be easily extended for approximate max-kernel search.

7.7.10.1 Absolute value approximation

From Theorem 11 and Algorithm 30, at any point in the single-tree algorithm with query point p_q and node $\mathcal{N}_i$ and best candidate kernel value $k^*(p_q)$, we know that we must descend $\mathcal{N}_i$ if

$$\mathcal{K}_{\max}(p_q, \mathcal{N}_i) \geq k^*(p_q) \tag{162}$$

but with absolute value approximation for some ϵ, we can loosen the condition to

[15]Here we are assuming that $\hat{k}(p_q) > 0$. In the case where $\hat{k}(p_q) < 0$, we seek a $p_r \in S_r$ such that $\mathcal{K}(p_q, p_r) > \hat{k}(p_q) - \epsilon|\hat{k}(p_q)|$

Algorithm 32 `Score(`p_q`,` $\mathscr{N}_r$`)` for absolute value approximation of FastMKS.

Input: query point p_q, reference space tree node $\mathscr{N}_r$, max-kernel candidate p^* for p_q and corresponding max-kernel value k^*, absolute value approximation ϵ
Output: a score for the node, or ∞ if the node can be pruned
if $\epsilon > \lambda_r \sqrt{\mathcal{K}(p_q, p_q)}$ **then**
    return ∞
else if $\mathcal{K}_{\max}(p_q, \mathscr{N}_r) < k^*$ **then**
    return ∞
else
    return $\mathcal{K}_{\max}(p_q, \mathscr{N}_r)$

$$\mathcal{K}_{\max}(p_q, \mathscr{N}_i) \geq k^*(p_q) + \epsilon \tag{163}$$

which can be simplified:

$$\begin{aligned} \mathcal{K}(p_q, p_i) + \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq k^*(p_q) + \epsilon \\ \mathcal{K}(p_q, p_i) + \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq \mathcal{K}(p_q, p_i) + \epsilon \\ \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq \epsilon. \end{aligned} \tag{164}$$

This yields that we can prune if $\epsilon > \lambda_i \sqrt{\mathcal{K}(p_q, p_q)}$. While this is looser than possible, it has the advantage that $\mathcal{K}(p_q, p_i)$ does not need to be calculated to prune $\mathscr{N}_i$. This yields a modified `Score()` algorithm, given in Algorithm 32.

In the dual-tree case, we must descend $(\mathscr{N}_q, \mathscr{N}_r)$ if

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) \geq B(\mathscr{N}_q). \tag{165}$$

Using absolute value approximation this condition loosens to

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) \geq B(\mathscr{N}_q) + \epsilon \tag{166}$$

but we cannot easily simplify this to eliminate the evaluation of $\mathcal{K}(p_q, p_r)$ due to the complexity of $B(\mathscr{N}_q)$. A modified `Score()` function for dual-tree absolute value approximate FastMKS is given in Algorithm 33.

Algorithm 33 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for absolute value approximation of FastMKS.

1: **Input:** query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$, absolute value approximation ϵ
2: **Output:** a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ or ∞ if the combination can be pruned
3: **if** $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) < B(\mathscr{N}_q) + \epsilon$ **then**
4: **return** ∞
5: **else**
6: **return** $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r)$

7.7.10.2 Relative value approximation

Relative value approximation is a more useful form of approximation, because the user does not need knowledge of $\hat{k}(p_q)$ to set ϵ reasonably. However, care has to be taken for relative value approximation because there is no guarantee that $\hat{k}(p_q) > 0$.

We can take Equation 162 and modify it for ϵ-relative-value-approximate pruning. In this case, we must descend $\mathscr{N}_i$ if

$$\mathcal{K}_{\max}(p_q, \mathscr{N}_i) \geq (1 + \epsilon)k^*(p_q) \tag{167}$$

and similar algebraic manipulations yield

$$\begin{aligned} \mathcal{K}(p_q, p_i) + \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq (1 + \epsilon)k^*(p_q) \\ \mathcal{K}(p_q, p_i) + \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq \mathcal{K}(p_q, p_i) + \epsilon k^*(p_q) \\ \lambda_i \sqrt{\mathcal{K}(p_q, p_q)} &\geq \epsilon k^*(p_q) \end{aligned}$$

meaning we can prune a node $\mathscr{N}_i$ when $k^*(p_q) > (\lambda_i/\epsilon)\sqrt{\mathcal{K}(p_q, p_q)}$. This is looser than possible (like the absolute-value approximation bound) but has the advantage that $\mathcal{K}(p_q, p_i)$ does not need to be calculated to prune $\mathscr{N}_i$. This yields a modified `Score()` algorithm, given in Algorithm 34.

Similar to absolute value approximation, we can loosen the condition for recursion given in Equation 165 to obtain the rule

Algorithm 34 `Score(`p_q`,` $\mathscr{N}_r$`)` for relative value approximation of FastMKS.

Input: query point p_q, reference space tree node $\mathscr{N}_r$, max-kernel candidate p^* for p_q and corresponding max-kernel value k^*, relative value approximation ϵ
Output: a score for the node, or ∞ if the node can be pruned
if $k^* > (\lambda_r / \epsilon) \sqrt{\mathcal{K}(p_q, p_q)}$ **then**
    return ∞
else if $\mathcal{K}_{\max}(p_q, \mathscr{N}_r) < k^*$ **then**
    return ∞
else
    return $\mathcal{K}_{\max}(p_q, \mathscr{N}_r)$

Algorithm 35 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for relative value approximation of FastMKS.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$, relative value approximation ϵ
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ or ∞ if the combination can be pruned
if $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) < (1 + \epsilon) B(\mathscr{N}_q)$ **then**
    return ∞
else
    return $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r)$

$$\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) \geq (1 + \epsilon) B(\mathscr{N}_q). \tag{168}$$

This rule does not easily simplify, as in the case of the single-tree relative value approximation rule; this is due to the complexity of $B(\mathscr{N}_q)$. A modified `Score()` function is given in Algorithm 35.

7.7.10.3 Rank Approximation

Rank approximation is a relatively new approximation paradigm introduced by Ram et al. [67]. The idea is to return a max-kernel candidate p'_r for query p_q, reference set S_r, and parameter τ such that p'_r is in the top τ max-kernel results with high probability. That is, for p_q, S_r, and τ, find an object $p_r \in S_r$ such that

$$\left| \left\{ p'_r \in S_r : \mathcal{K}(p_q, p'_r) > \mathcal{K}(q, p_r) \right\} \right| < \tau. \tag{169}$$

This is often a better technique than absolute-value-approximate search, which requires

Algorithm 36 `Score`$(p_q,\ \mathscr{N}_r)$ for rank approximation of FastMKS.

Input: query point p_q, reference space tree node $\mathscr{N}_r$, max-kernel candidate p^* for p_q and corresponding max-kernel value k^*, required number of samples k for τ-rank approximation in a reference set of size n
Output: a score for the node, or ∞ if the node can be pruned
if $|\mathscr{D}_r^p| \leq (n/k)$ **then**
  $S'_r \leftarrow \lceil (k/n)|\mathscr{D}_r^p| \rceil$ random samples from $|\mathscr{D}_r^p|$
  for all $p'_r \in S'_r$ **do**
    `BaseCase`$(p_q,\ p'_r)$
  return ∞
else if $\mathcal{K}_{\max}(p_q, \mathscr{N}_r) < k^*$ **then**
  return ∞
else
  return $\mathcal{K}_{\max}(p_q, \mathscr{N}_r)$

a tuned parameter ϵ for each dataset, and relative-value-approximate search, which may return useless results when the values of $\mathcal{K}(p_q, p_r)$ are very close for all $p_r \in S_r$.

The idea presented in [67] is to draw a set of samples S'_r large enough that the maximum kernel value between p_q and any point in S'_r (call this k^*) is such that

$$\Pr\left(\left|\left\{p'_r \in S_r : \mathcal{K}(p_q, p'_r) > k^*\right\}\right| < \tau\right) \geq 1 - \delta. \tag{170}$$

Simplifying the formulation presented in [67], the probability of always missing the top τ values for a given query p_q after k samples with replacement is given by $(1 - (\tau/n))^k$, where $|S_r| = n$. If we want a $(1 - \delta)$ success rate of sampling, then we want k to be such that

$$\left(1 - \frac{\tau}{n}\right)^k < \delta, \quad \text{and} \quad \left(1 - \frac{\tau}{n}\right)^{k-1} > \delta, \tag{171}$$

which gives

$$k = \left\lceil \frac{\log \delta}{\log\left(1 - \frac{\tau}{n}\right)} \right\rceil. \tag{172}$$

Following the logic of [67], if a node $\mathscr{N}_i$ contains more than (n/k) points ($|\mathscr{D}_i^p| > (n/k)$), then we can prune the node *after* we randomly sample $\lceil (k/n)|\mathscr{D}_i^p| \rceil$ points from $|\mathscr{D}_i^p|$. In

Algorithm 37 `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` for rank approximation of FastMKS.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$, required number of samples k for τ rank approximation in a reference set of size n
Output: a score for the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ or ∞ if the combination can be pruned
if $|\mathscr{D}_r^p| \leq (n/k)$ **then**
  for all $p_q \in \mathscr{D}_q^p$ **do**
    $S'_r \leftarrow \lceil (k/n)|\mathscr{D}_r^p| \rceil$ random samples from $|\mathscr{D}_r^p|$
    for all $p'_r \in S'_r$ **do**
      `BaseCase(`p_q`,` p'_r`)`
  return ∞
else if $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r) > B(\mathscr{N}_q)$ **then**
  return ∞
else
  return $\mathcal{K}_{\max}(\mathscr{N}_q, \mathscr{N}_r)$

addition to that, the standard FastMKS pruning rules still apply. An updated single-tree `Score()` function is given in Algorithm 36.

An extension of this for a dual-tree algorithm is straightforward; for a reference node $\mathscr{N}_r$, if $|\mathscr{D}_r^p| > (n/k)$, then we can sample it for each query point $p_q \in \mathscr{D}_q^p$ and prune the node combination. A `Score()` function is given in Algorithm 37.

7.7.11 Empirical evaluation

We evaluate single-tree and dual-tree FastMKS with different kernels and datasets. For each experiment, we query the top $\{1, 2, 5, 10\}$ max-kernel candidates and report the speedup over linear search (in terms of the number of kernel evaluations performed during the search). The cover tree and the algorithms are implemented in C++ in **mlpack** [87].

7.7.11.1 Datasets

We use two different classes of datasets. First, we use datasets with fixed-length objects. These include the MNIST dataset [138], the Isomap "Images" dataset, several datasets from the UCI machine learning repository [134], three collaborative filtering datasets (MovieLens, Netflix [214], Yahoo! Music [202]), the LCDM astronomy dataset [147], the LiveJournal blog moods text dataset [215] and a subset of the 80 Million Tiny Images dataset

Table 20: Vector dataset details; $|S_q|$ and $|S_r|$ denote the number of objects in the query and reference sets respectively and *dims* denotes the dimensionality of the sets.

Datasets	$\|S_q\|$	$\|S_r\|$	*dims*
Y! Music	10000	624961	51
MovieLens	6040	3706	11
Opt-digits	450	1347	64
Physics	37500	112500	78
Homology	75000	210409	74
Covertype	100000	481012	55
LiveJournal	10000	10000	25327
MNIST	10000	60000	784
Netflix	17770	480189	51
Corel	10000	27749	32
LCDM	6000000	10777216	3
TinyImages	1000	1000000	384

[216]. The sizes of the datasets are presented in Table 20.

The second class of dataset we use are those without fixed length representation. We use protein sequences from GenBank[16].

7.7.11.2 Kernels

We consider the following kernels for the vector datasets:

- linear: $\mathcal{K}(x, y) = x^T y$
- polynomial: $\mathcal{K}(x, y) = (x^T y)^2$
- cosine: $\mathcal{K}(x, y) = (x^T y)/(\|x^T\|\|y\|)$
- polynomial, deg. 10: $\mathcal{K}(x, y) = (x^T y)^{10}$
- Epanechnikov: $\mathcal{K}(x, y) = \max(0, 1 - \|x - y\|^2/b^2)$

While the Epanechnikov kernel is normalized and thus reduces to nearest neighbor search, we choose it regardless to show the applicability of FastMKS to a variety of kernels. It is important to remember that standard techniques for nearest neighbor search

[16]See `ftp://ftp.ncbi.nih.gov/refseq/release/complete`.

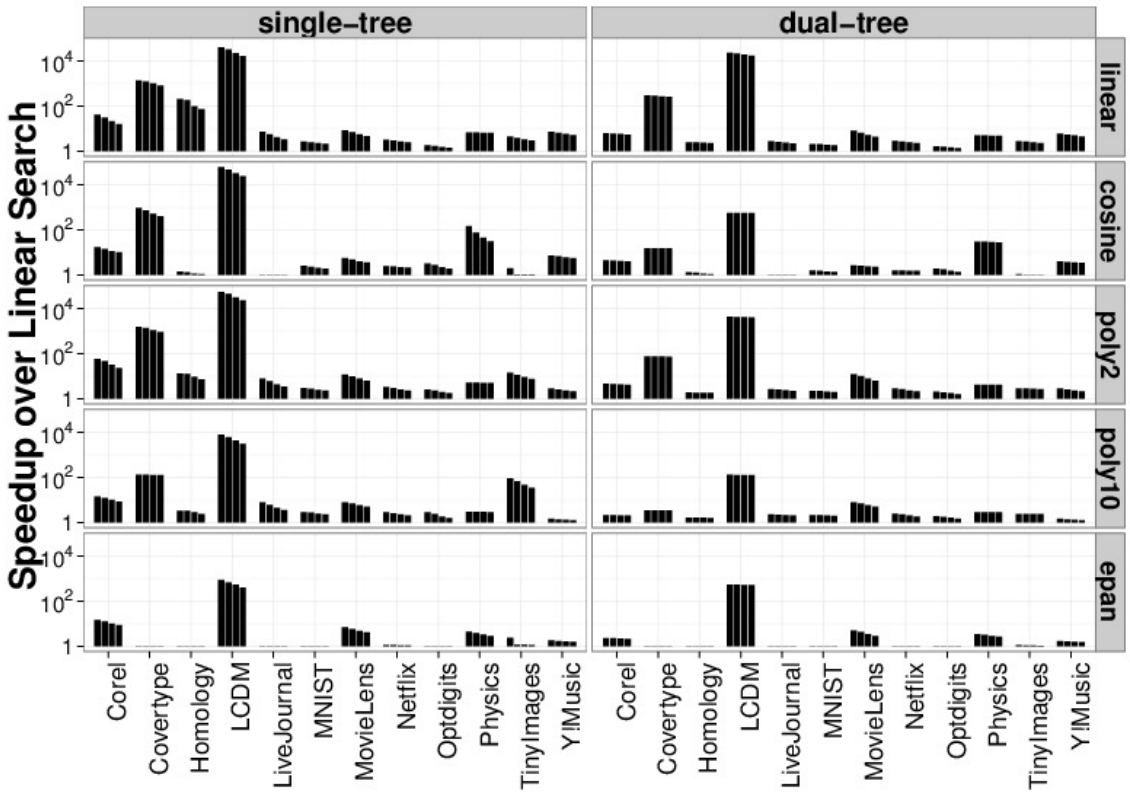


Figure 39: Speedups of single-tree and dual-tree FastMKS over linear scan with $k = \{1, 2, 5, 10\}$.

should be able to perform the task faster—we do not compare with those techniques in these experiments.

For the protein sequences, we use the p-spectrum string kernel [198], which is a measure of string similarity. The kernel value for two given strings is the number of length-p substrings that appear in both strings.

7.7.11.3 Implementation

For maximum performance, the implementation in **mlpack** does not precisely follow the algorithms we have given. By default, the cover tree is designed to use a base of 2 during construction, but following the authors' observations, we find that a base of 1.3 seems to give better performance results [57]. In addition, for both the single-tree algorithms, we attempt to first score nodes (and node combinations) whose kernel values $\mathcal{K}(p_q, p_r)$ are higher, in hopes of tightening the bounds $B(\mathcal{N}_q)$ and $k^*(p_q)$ more quickly.

Lastly, the `Score()` method as implemented in **mlpack** is somewhat more complex: it attempts to prune the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ with a looser bound that does not evaluate $\mathcal{K}(p_q, p_r)$. If that is not successful, `Score()` proceeds as in Algorithm 31 (or 30 in the single-tree case). This type of prune seems to give 10–30% reductions in the number of kernel evaluations (or more, depending on the dataset).

The **mlpack** implementation can be downloaded from `http://www.mlpack.org/` and its FastMKS implementation includes both C++ library bindings for FastMKS and each kernel we have discussed as well as a `fastmks` executable that can be used to run FastMKS easily from the command line. In addition, a tutorial can be found on the website, and the source code is extensively documented.

7.7.11.4 Results

The results for the vector datasets are summarized in Figure 39 and detailed for $k = 1$ in Tables 21 and 22. The tables also provide the number of kernel evaluations calculated during the search for linear search, single-tree FastMKS, and dual-tree FastMKS. Speedups over a factor of 100 are highlighted in bold. While the speedups range from anywhere between 1 (which indicates no speedup) to 50000, many datasets give speedups of an order of magnitude or more. As would be expected with the postulated $O(\log N)$ bounds for single-tree FastMKS and the $O(N)$ bounds for dual-tree FastMKS, larger datasets (such as LCDM) tend to provide larger speedups. In the cases where large datasets are used but small speedup values are obtained, the conclusion must be that the expansion constant c_r and the directional concentration constant γ_r for that dataset and kernel are large. In addition, the Epanechnikov kernel is parameterized by a bandwidth b; this bandwidth will seriously affect the runtime if it is too small (all kernel evaluations are 0) or too large (all kernel evaluations are 1). We have arbitrarily chosen 10 as our bandwidth for simplicity in simulations, but for each dataset, it is certain that a better bandwidth value that will provide additional speedup exists.

Another observation is that the single-tree algorithm tends to perform better than the

Table 21: Single-tree and dual-tree FastMKS on vector datasets with $k = 1$, part one.

		Kernel evaluations			**Speedup**	
Kernel	Dataset	Linear scan	Single-tree	Dual-tree	Single-tree	Dual-tree
linear	Y! Music	6.249B	859.1M	1.056B	7.27	5.91
	MovieLens	22.38M	2.635M	2.790M	8.49	8.02
	Optdigits	606.1k	333.2k	366.6k	1.82	1.65
	Physics	4.219B	628.8M	852.9M	6.71	4.95
	Bio	20.36B	100.2M	8.174B	**203.2**	2.49
	Covertype	48.10B	35.06M	160.9M	**1372**	**299.0**
	LiveJournal	100.0M	13.88M	36.09M	7.21	2.77
	MNIST	600.0M	229.6M	288.2M	2.62	2.08
	Netflix	8.532B	2.632B	2.979B	3.12	2.86
	Corel	277.5M	6.626M	44.02M	41.88	6.30
	LCDM	64.66T	1.566B	2.778B	**41282**	**23269**
	TinyImages	100.0M	22.30M	35.70M	4.48	2.80
poly.	Y! Music	6.249B	2.187B	2.221B	2.86	2.81
	MovieLens	22.38M	1.865M	1.833M	12.00	12.21
	Optdigits	606.1k	235.1k	296.5k	2.58	2.04
	Physics	4.219B	823.9M	1.017B	5.12	4.15
	Bio	20.36B	1.538B	10.87B	13.23	1.87
	Covertype	48.10B	30.65M	629.7M	**1569**	76.39
	LiveJournal	100.0M	12.91M	38.16M	7.75	2.62
	MNIST	600.0M	202.8M	266.8M	2.96	2.25
	Netflix	8.532B	2.528B	2.953B	3.37	2.89
	Corel	277.5M	4.687M	60.30M	59.20	4.60
	LCDM	64.66T	1.171B	14.98B	**55204**	**4316**
	TinyImages	100.0M	6.957M	34.32M	14.37	2.91
poly.	Y! Music	6.249B	4.296B	4.310B	1.45	1.45
deg. 10	MovieLens	22.38M	2.814M	2.826M	7.96	7.92
	Optdigits	606.1k	212.3k	318.2k	2.86	1.91
	Physics	4.219B	1.441B	1.481B	2.93	2.91
	Bio	20.36B	6.018B	12.45B	3.38	1.63
	Covertype	48.10B	361.1M	13.78B	**133.2**	3.49
	LiveJournal	100.0M	12.75M	43.25M	7.84	2.31
	MNIST	600.0M	205.4M	277.1M	2.92	2.17
	Netflix	8.532B	2.977B	3.470B	2.87	2.46
	Corel	277.5M	19.68M	131.1M	14.10	2.12
	LCDM	64.66T	8.124B	485.2B	**7959**	**133.3**
	TinyImages	100.0M	1.076M	42.23M	92.96	2.37

dual-tree algorithm, in spite of the better scaling of the dual-tree algorithm. There are multiple potential explanations for this phenomenon:

Table 22: Single-tree and dual-tree FastMKS on vector datasets with $k = 1$, part two.

		Kernel evaluations			**Speedup**	
Kernel	Dataset	Linear scan	Single-tree	Dual-tree	Single-tree	Dual-tree
cosine	Y! Music	6.249B	849.6M	1.586B	7.36	3.94
	MovieLens	22.38M	4.044M	8.322M	5.54	2.69
	Optdigits	606.1k	190.0k	319.8k	3.19	1.90
	Physics	4.219B	28.82M	140.0M	**146.3**	30.14
	Bio	20.36B	14.40B	15.54B	1.41	1.31
	Covertype	48.10B	50.15M	3.119B	**959.2**	15.42
	LiveJournal	100.0M	99.23M	98.78M	1.01	1.01
	MNIST	600.0M	237.0M	376.7M	2.53	1.59
	Netflix	8.532B	3.426B	5.344B	2.49	1.60
	Corel	277.5M	16.22M	61.95M	17.10	4.48
	LCDM	64.66T	1.058B	112.9B	**61063**	**572.6**
	TinyImages	100.0M	50.49M	92.36M	1.98	1.02
Epan.	Y! Music	6.249B	3.439B	3.630B	1.82	1.72
	MovieLens	22.38M	3.243M	4.471M	6.90	5.01
	Optdigits	606.1k	606.1k	606.1k	1.00	1.00
	Physics	4.219B	957.6M	1.213B	4.40	3.48
	Bio	20.36B	20.25B	20.25B	1.01	1.01
	Covertype	48.10B	48.10B	48.10B	1.00	1.00
	LiveJournal	100.0M	99.57M	99.15M	1.00	1.01
	MNIST	600.0M	600.0M	600.0M	1.00	1.00
	Netflix	8.532B	7.602B	8.293B	1.12	1.03
	Corel	277.5M	18.53M	119.9M	14.98	2.31
	LCDM	64.66T	72.32B	119.0B	**894.1**	**543.3**
	TinyImages	100.0M	42.49M	87.99M	2.35	1.14

- The single-tree bounds given in Theorem 11 (Equation 103) and Theorem 13 (Equation 113) are tighter than the dual-tree bounds of Theorem 12 (Equation 107) and Theorem 14 (Equation 121).

- The dual-tree algorithm's runtime is also bounded by the parameters ν, η_r, and τ_q, whereas the single-tree algorithm is not. This could mean that N would need to be very large before the dual-tree algorithm became faster, despite the fact that the dual-tree algorithm scales with c_r^7 and the single-tree algorithm scales with c_r^{12}.

- The single-tree algorithm scales considers each element in the set $|S_q|$ linearly, but the dual-tree algorithm is able to obtain max-kernel bounds for many query points at

Table 23: Single-tree and dual-tree FastMKS on protein sequences with $k = 1$.

		Kernel evaluations			**Speedup**	
$\|S_q\|$	$\|S_r\|$	Linear scan	Single-tree	Dual-tree	Single-tree	Dual-tree
391	649	253.8k	5.255k	43.27k	48.29	5.87
1091	649	708.1k	14.99k	122.7k	47.25	5.77
2635	649	1.710M	36.04k	327.7k	47.45	5.22
8604	649	5.584M	115.3k	832.9k	48.43	6.70
37606	649	24.41M	512.9k	3.763M	47.58	6.49
63180	649	41.00M	848.1k	4.999M	48.35	8.20
63180	391	24.70M	484.3k	3.511M	51.01	7.04
63180	1091	68.93M	834.9k	7.529M	82.56	9.16
63180	2635	166.5M	927.8k	22.95M	**179.4**	7.25
63180	8604	543.6M	692.5k	32.26M	**785.1**	16.85
63180	37606	2.376B	743.2k	65.09M	**3197**	36.50
63180	63180	3.992B	1.140M	150.2M	**3500**	26.59
391	391	152.8k	2.973k	30.68k	51.42	4.98
1091	1091	1.190M	14.96k	183.2k	79.56	6.50
2635	2635	6.943M	43.95k	1.689M	**158.0**	4.11
8604	8604	323.6M	104.4k	13.76M	**783.2**	12.79
37606	37606	1.414B	470.2k	39.70M	**3007**	35.62
63180	63180	3.992B	1.141M	150.2M	**3500**	26.59

once thanks to the use of the second tree. Thus, the dual-tree algorithm may require a much larger S_q before it outperforms the single-tree algorithm.

The results for the protein sequence data are shown in Figure 40 and Table 23. The table shows that for constant reference set size (649), the dual-tree algorithm provides better scaling as the query set grows. This agrees with the better scaling of dual-tree FastMKS as exhibited in Theorem 17.

However, in every case in Table 23, the single-tree algorithm provides better performance than the dual-tree algorithm. This implies that the query sets and reference sets would have to be possibly several orders of magnitude larger for the dual-tree algorithm to provide better speedups. With larger datasets, the single-tree algorithm showed more than 3000x speedup over linear scan. Other datasets may exhibit better or worse scaling depending on the expansion constant and directional concentration constant.

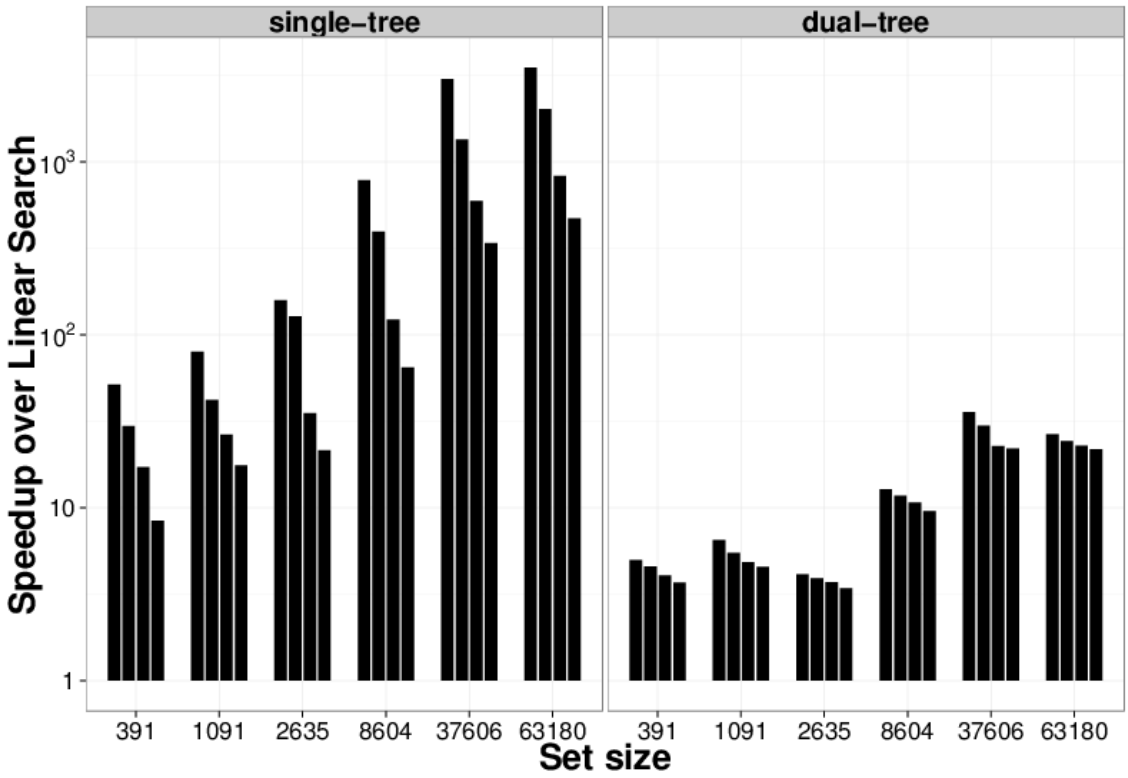


Figure 40: Speedups of single-tree and dual-tree FastMKS over linear scan for protein sequences with $k = \{1, 2, 5, 10\}$.

7.7.12 Future directions for max-kernel search

We have now presented two algorithms to solve exact max-kernel search, and six additional extensions for approximation. However, as always, there is room for improvement and further investigation, and a few avenues are laid out below. Another possible avenue is parallelism, but this is an improvement to a traversal, and thus is not restricted to max-kernel search. Traversal improvements are detailed further in Chapter 6.

7.7.12.1 Tighter bounds for specific kernels

In Theorems 13 and 14 we described a tighter bound for normalized kernels ($\mathcal{K}(x, x) = 1 \; \forall x$). It is our intuition that similar tighter bounds can be developed for other specific types of Mercer kernels.

This may be especially applicable in domain-specific kernels such as string kernels or graph kernels. Any kernel that has some known structure on how points are mapped to $\mathcal{H}$

may be bounded more tightly than the general Mercer kernel bounds given in Equations 103 and 107.

7.7.12.2 Domain-specific applications

In the introduction, we mentioned the wide applicability of max-kernel search, discussing its use in image retrieval, document retrieval, collaborative filtering, and even finding similar protein/DNA sequences. That list only contains a few of the numerous max-kernel search problems that arise ubiquitiously in countless fields (not just computing-related fields).

In many of these fields, there are existing domain-specific solutions. One example in genomics is BLAST (Basic Local Alignment Search Tool) [203], a utility that searches for similarity between biological sequences. Another tool of this sort is the older FASTA algorithm [217]. Both of these algorithms are improvements over linear scan with the Smith-Waterman alignment score [204]. However, in contrast with its large speedups, BLAST cannot guarantee exact results.

The Smith-Waterman alignment score can easily be shown to be a Mercer kernel; therefore, FastMKS could be used to give speedups over linear scan and it would also return provably exact results. Furthermore, approximation extensions to FastMKS could provide additional speedups by relaxing the exact result constraint, potentially making FastMKS competitive with BLAST.

7.7.13 Wrap-up for max-kernel search

In this section, we have described a tree-independent single-tree and dual-tree algorithm in Algorithms 29, 30 and 31 which are able to quickly perform the task of max-kernel search, for individual query points and also for sizeable query sets. As we have seen, max-kernel search is ubiquitious in computer science, so these algorithms—the first to solve max-kernel search exactly—are groundbreaking and useful. In addition, the dual-tree algorithm can be shown to scale linearly, with some assumptions on the dataset. Lastly, the empirical

results show both algorithms to be effective in practice.

Next, we will move toward another common problem in machine learning—clustering.

7.8 k-means clustering

This section develops a dual-tree algorithm for fast k-means clustering for large datasets and large k. It is based upon work recently submitted [80].

7.8.1 Introduction

Of all the clustering algorithms in use today, among the simplest and most utilized is the venerated k-means clustering algorithm, usually implemented via Lloyd's algorithm. Lloyd's algorithm is quite simple and well-known: given a dataset S, repeat the following two steps (a 'Lloyd iteration') until the centroids of each of the k clusters converge:

1. Assign each point $p_i \in S$ to the cluster with nearest centroid.
2. Recalculate the centroids for each cluster using the assignments of each point in S.

Clearly, a simple implementation of this algorithm will take $O(kN)$ time where $N = |S|$. However, the number of iterations is not bounded unless the practitioner manually sets a maximum, and k-means is not guaranteed to converge to the global best clustering. Despite these shortcomings, in practice k-means tends to quickly converge to reasonable solutions. Even so, there is no shortage of techniques for improving the clusters k-means converges to: refinement of initial centroid selection [218] and weighted sampling of initial centroids [219] are just two of the many popular existing strategies.

There are also a number of methods for accelerating the runtime of a single iteration of k-means. In general, these ideas use the triangle inequality to prune work during the assignments step. Pelleg and Moore [39] build a kd-tree on the set S in order to rule out certain centroids for entire kd-tree nodes. Elkan [220] describes an algorithm which constructs an $O(k^2 + kN)$-size data structure to store between-cluster distances and bounds; Hamerly [221] proposes his own improvement to Elkan's algorithm. These algorithms have

been shown to provide massive speedup—however, the quadratic scaling in k of Hamerly and Elkan's algorithms makes them problematic for large k.

In the development of this fast dual-tree k-means algorithm, we first show the relevance of the large k case; then, we observe that a tree can also be built on the k clusters, and then a dual-tree algorithm [28, 61] can be used to efficiently perform an exact single iteration of k-means clustering. The dual-tree algorithm developed here is independent of the type of tree used and therefore can be used with not only *kd*-trees, but also metric trees, cover trees, and other types of space trees [28]. When cover trees are used, we use adaptive runtime analysis techniques to show a worst-case runtime bound of $O(N + k \log k)$; this bound depends on properties of the dataset. To our knowledge, these are the first sub-$O(kN)$ worst-case runtime bounds for an exact Lloyd iteration. Empirical results indicate that our algorithm is the best in its intended scenario: the large k and large N case.

7.8.2 Scaling k-means

Although the original publications on k-means only applied the algorithm to a maximum dataset size of 760 points, the half-century of relentless progress since then has seen dataset sizes scale into the billions. Due to its simplicity, though, k-means has remained relevant, and is still applied in numerous large-scale applications.

In cases where N scales but k remains small, a good choice of algorithm is a sampling algorithm, which will return an approximate clustering. One sampling technique, coresets, can produce good clusterings for n in the millions using several hundred or a few thousand points [222]. However, for large k, the number of samples required to produce good clusterings can become prohibitive.

For large k, then, we turn to an alternative approach: accelerating exact Lloyd iterations. Existing techniques include the naive linear scan implementation implied in the previous section, the *blacklist* algorithm [39], Elkan's algorithm [220], and Hamerly's algorithm [221]. The blacklist algorithm builds a *kd*-tree on the dataset and, while the tree is traversed, blacklists individual clusters that cannot be the closest cluster (the *owner*) of

Table 24: Runtime and memory bounds for k-means algorithms.

Algorithm	**Setup**	**Worst-case**	**Memory**
naive	n/a	$O(kN)$	$O(k+N)$
blacklist	$O(N \log N)$	$O(kN)$	$O(k \log N + N)$
elkan	n/a	$O(k^2 + kN)$	$O(k^2 + kN)$
hamerly	n/a	$O(k^2 + kN)$	$O(k+N)$
dualtree	$O(N \log N)$	$O(k \log k + N)$[1]	$O(k+N)$

any descendant points of a node. Elkan's algorithm maintains an upper bound and a lower bound on the distance between each point and centroid; Hamerly's algorithm is a simplification of this technique which uses less memory. Table 24 shows setup costs, worst-case per-iteration runtimes, and memory usage of each of these algorithms as well as the proposed dual-tree algorithm[17]. The expected runtime of the blacklist algorithm is, under some assumptions, roughly $O(k + k \log N + N)$ per iteration. The expected runtime of Hamerly's and Elkan's algorithm is $O(k^2 + \alpha N)$ time, where α is the expected number of clusters visited by each point (in both Elkan and Hamerly's results, α seems to be small).

However, none of these algorithms are specifically tailored to the large k case, and the large k case is common. Pelleg and Moore [39] report several hundred clusters in a subset of 800k objects from the SDSS dataset. Clusterings for n-body simulations on astronomical data often involve several thousand clusters [223]. Csurka et al. [224] extract vocabularies from image sets using k-means with $k \sim 1000$. Coates et al. [225] show that k-means can work surprisingly well for unsupervised feature learning for images, using k as large as 4000 on 50000 images. Also, in text mining, datasets can have up to 18000 unique labels [226]. Can and Ozkarahan [227] suggest that the number of clusters in text data is directly related to the size of the vocabulary, suggesting $k \sim mN/t$ where m is the vocabulary size, n is the number of documents, and t is the number of nonzero entries in the term matrix. Further, in vector quantization codebook generation, for which k-means is sometimes used, k may be in the tens of thousands [225].

[17]The worst-case runtime bound for the dual-tree algorithm also depends on some assumptions on dataset-dependent constants. This is detailed further in Section 7.5.6.

Thus, it is important to have an algorithm with favorable scaling properties for both large k and N.

7.8.3 The blacklist algorithm and trees

The blacklist algorithm is a single-tree algorithm: one tree (the reference tree) is built on the dataset, and then that tree is traversed in order to assign each point to its nearest cluster centroid. We know from Chapter 2 that there are numerous single-tree algorithms to solve a whole host of problems, and we also know that most of these single-tree algorithms have related dual-tree algorithms.

Following the empirical success of the blacklist algorithm, then, it is only natural to build a tree on the data points. Tree-building is (generally) a one-time $O(N \log N)$ cost and for large N and/or k, the cost of tree building is often negligible compared to the time it takes to perform the clustering. We may also build a tree on the k centroids (the query tree) which will allow us to rule out *many* centroids for *many* points at once.

Due to the tree-independent dual-tree algorithm abstraction introduced in Chapter 3, we know that to describe a dual-tree algorithm we only need to provide a `BaseCase()` and `Score()` function. Then, we may use any tree and any traversal in order to create a working dual-tree algorithm.

The two types of trees we will explicitly consider for dual-tree k-means are the *kd*-tree [31] and the cover tree [57], but it should be remembered that the algorithm as provided is sufficiently general to work with any other type of tree. As with everything in this thesis, notation is standardized according to Section 3.4 and Table 1.

7.8.4 Pruning strategies

All of the existing accelerated k-means algorithms operate by avoiding unnecessary work via the use of pruning strategies.

A first observation is that the first step of a Lloyd iteration—assign each point $p_i \in S$ to the cluster with the nearest centroid—is exactly nearest neighbor search, with the set of

query points equal to S and the set of reference points equal to the set of centroids. Thus, it is possible to use dual-tree nearest neighbor search as a black box. However, we may implement more complex pruning strategies which can give significantly more speedup. We will base our algorithm on dual-tree nearest neighbor search with the S as the query points and the centroids as the reference points. Below are the four pruning strategies we will pursue:

Strategy one. When visiting a particular combination $(\mathscr{N}_q, \mathscr{N}_r)$ (with $\mathscr{N}_q$ holding points in the dataset and $\mathscr{N}_r$ holding centroids), the combination should be pruned if every descendant centroid in $\mathscr{N}_r$ can be shown to own none of the points in $\mathscr{N}_q$. If we have cached an upper bound $\text{ub}(\mathscr{N}_q)$ on the distance between any descendant point of $\mathscr{N}_q$ and its nearest cluster centroid that satisfies

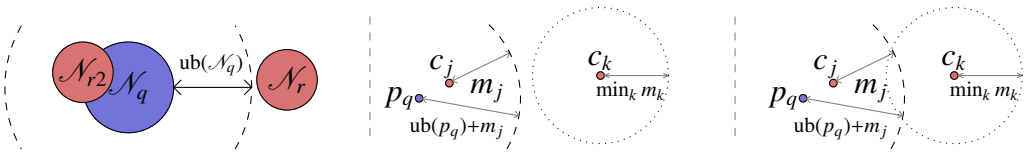
$$\text{ub}(\mathscr{N}_q) \geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_q) \tag{173}$$

where c_q is the cluster centroid nearest to point p_q, the node $\mathscr{N}_r$ can contain no centroids that own any descendant points of $\mathscr{N}_q$ if

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > \text{ub}(\mathscr{N}_q). \tag{174}$$

This relation bears similarity to the pruning rules for nearest neighbor search [28] and max-kernel search [79]. A graphical depiction of a situation where $\mathscr{N}_r$ can be pruned is given in Figure 41a; in this case, ball-shaped tree nodes are used, and the upper bound $\text{ub}(\mathscr{N}_q)$ is set to $d_{\max}(\mathscr{N}_q, \mathscr{N}_{r2})$.

Strategy two. The recursion down a particular branch of the query tree should terminate early if we can determine that only one cluster can possibly own all of the descendant points of that branch. This is related to the first strategy. If we have been caching the number of pruned centroids (call this $\text{pruned}(\mathscr{N}_q)$), as well as the identity of any arbitrary non-pruned centroid (call this $\text{closest}(\mathscr{N}_q)$), then if $\text{pruned}(\mathscr{N}_q) = k - 1$, we may conclude that the centroid $\text{closest}(\mathscr{N}_q)$ is the owner of all descendant points of $\mathscr{N}_q$, and there is no



(a) $\mathcal{N}_r$ can be pruned. (b) p_q's owner cannot change. (c) p_q's owner can change.

Figure 41: Different pruning situations.

need for further recursion in $\mathcal{N}_q$.

Strategy three. The traversal should not visit nodes whose owner could not have possibly changed between iterations; that is, the tree should be coalesced before traversal to include only nodes whose owners may have changed.

There are two easy ways to use the triangle inequality to show that the owner of a point cannot change between iterations. Figures 41b and 41c show the first: assume that we have a point p_q with owner c_j and second-closest centroid c_k. Between iterations, each centroid will move when it is recalculated; define the distance that centroid c_i has moved as m_i. Using these quantities, we may bound the distances for the next iteration: $d(p_q, c_j)+m_j$ is an upper bound on the distance between p_q and its owner next iteration, and $d(p_q, c_k)-\max_i m_i$ is a lower bound on the distance between p_q and its second closest centroid next iteration. We may use these bounds to conclude that if

$$d(p_q, c_j) + m_j < d(p_q, c_k) - \max_i m_i, \tag{175}$$

then the owner of p_q next iteration must be c_j. Now, let us generalize this from individual points p_q to tree nodes $\mathcal{N}_q$. This pruning strategy can only be used when all descendant points of $\mathcal{N}_q$ are owned by a single centroid, and in order to perform the prune, we need to establish a lower bound on the distance between any descendant point of the node $\mathcal{N}_q$ and the second closest centroid. Call this bound lb($\mathcal{N}_q$). Now, remember that ub($\mathcal{N}_q$) provides an upper bound on the distance between any descendant point of $\mathcal{N}_q$ and its nearest centroid. Then, if all descendant points of $\mathcal{N}_q$ are owned by some cluster c_j in one iteration, and

$$\text{ub}(\mathcal{N}_q) + m_j < \text{lb}(\mathcal{N}_q) - \max_i m_i, \tag{176}$$

then $\mathcal{N}_q$ is owned by cluster c_j in the next iteration. Implementationally, it is convenient to have $\text{lb}(\mathcal{N}_q)$ store a lower bound on the distance between any descendant point of $\mathcal{N}_q$ and the nearest pruned centroid. Then, if $\mathcal{N}_r$ is entirely owned by one cluster, all other centroids are pruned, and $\text{lb}(\mathcal{N}_q)$ holds the necessary lower bound for pruning according to the rule above.

The second way to use the triangle inequality to show that an owner cannot change depends on the distances between centroids. Suppose that p_q is owned by c_j at the current iteration; then, if

$$d(p_q, c_j) - m_j < 2\left(\min_{c_i \in C, c_i \neq c_j} d(c_i, c_j)\right) \tag{177}$$

then c_j will own p_q next iteration [220]. We may adapt this rule to tree nodes $\mathcal{N}_q$ in the same way as the previous rule; if $\mathcal{N}_q$ is owned by cluster c_j during this iteration and

$$\text{ub}(\mathcal{N}_q) + m_j < 2\left(\min_{c_i \in C, c_i \neq c_j} d(c_i, c_j)\right) \tag{178}$$

then $\mathcal{N}_q$ is owned by cluster c_j in the next iteration. Note that the above rules do work with individual points p_q instead of nodes $\mathcal{N}_q$ if we have a valid upper bound $\text{ub}(p_q)$ and a valid lower bound $\text{lb}(p_q)$. Any nodes or points that satisfy the above conditions do not need to be visited during the next iteration, and thus can be removed from the tree for the next iteration.

Strategy four. The traversal should use bounding information from previous iterations; for instance, $\text{ub}(\mathcal{N}_q)$ should not be reset to ∞ at the beginning of each iteration. Between iterations, we may update $\text{ub}(\mathcal{N}_q)$, $\text{ub}(p_q)$, $\text{lb}(\mathcal{N}_q)$, and $\text{lb}(p_q)$ according to the following rules:

$$\text{ub}(\mathscr{N}_q) \leftarrow \begin{cases} \text{ub}(\mathscr{N}_q) + m_j & \text{if } \mathscr{N}_q \text{ is owned by a single cluster } c_j \\ \text{ub}(\mathscr{N}_q) + \max_i m_i & \text{if } \mathscr{N}_q \text{ is not owned by a single cluster,} \end{cases} \tag{179}$$

$$\text{ub}(p_q) \leftarrow \text{ub}(p_q) + m_j, \tag{180}$$

$$\text{lb}(\mathscr{N}_q) \leftarrow \text{lb}(\mathscr{N}_q) - \max_i m_i, \tag{181}$$

$$\text{lb}(p_q) \leftarrow \text{lb}(p_q) - \max_i m_i. \tag{182}$$

Note that special handling is required when descendant points of $\mathscr{N}_q$ are not owned by a single centroid (Equation 179). It is also true that for a child node $\mathscr{N}_c$ of $\mathscr{N}_q$, $\text{ub}(\mathscr{N}_q)$ is a valid upper bound for $\mathscr{N}_c$ and $\text{lb}(\mathscr{N}_q)$ is a valid lower bound for $\mathscr{N}_c$: that is, the upper and lower bounds may be taken from a parent, and they are still valid.

7.8.5 The dual-tree k-means algorithm

These four pruning strategies lead to a high-level k-means algorithm, described in Algorithm 38. During the course of this algorithm, to implement each of our pruning strategies, we will need to maintain the following quantities:

- $\text{ub}(\mathscr{N}_q)$: an upper bound on the distance between any descendant point of a node $\mathscr{N}_q$ and the nearest centroid to that point.
- $\text{lb}(\mathscr{N}_q)$: a lower bound on the distance between any descendant point of a node $\mathscr{N}_q$ and the nearest pruned centroid.
- $\text{pruned}(\mathscr{N}_q)$: the number of centroids pruned during traversal for $\mathscr{N}_q$.
- $\text{closest}(\mathscr{N}_q)$: if $\text{pruned}(\mathscr{N}_q) = k - 1$, this holds the owner of all descendant points of $\mathscr{N}_q$.
- $\text{canchange}(\mathscr{N}_q)$: whether or not $\mathscr{N}_q$ can change owners next iteration.
- $\text{ub}(p_q)$: an upper bound on the distance between point p_q and its nearest centroid.

Algorithm 38 High-level outline of dual-tree k-means.

```
1: Input: dataset S ∈ R^{N×d}, initial centroids C ∈ R^{k×d}.
2: Output: converged centroids C.
3: T ← a tree built on S
4: while centroids C not converged do
5:    {Remove nodes in the tree whose nearest cluster is already known.}
6:    T ← CoalesceNodes(T)
7:    T_c ← a tree built on C
8:    {Call dual-tree algorithm for finding nearest clusters.}
9:    Perform a dual-tree recursion with T, T_c, BaseCase(), and Score().
10:   {Restore the tree to its non-coalesced form.}
11:   T ← DecoalesceNodes(T)
12:   {Update centroids and bounding information in the tree.}
13:   C ← UpdateCentroids(T)
14:   T ← UpdateTree(T)
15: return C
```

- lb(p_q): a lower bound on the distance between point p_q and its second nearest centroid.
- closest(p_q): the closest centroid to p_q (this is also the owner of p_q).
- canchange(p_q): whether or not p_q can change owners next iteration.

At the beginning of the algorithm, each upper bound is initialized to ∞, each lower bound is initialized to ∞, pruned($\cdot$) is initialized to 0 for each node, and closest($\cdot$) is initialized to an invalid centroid for each cluster and point. canchange($\cdot$) is initialized to `true` for each node and point. Because of this, line 6 will do nothing on the first iteration.

7.8.5.1 `BaseCase()` *and* `Score()`

First, consider the dual-tree algorithm called on line 9. As detailed earlier, we can describe a dual-tree algorithm as a combination of tree type, traversal type, and point-to-point `BaseCase()` and node-to-node `Score()` functions. Therefore, we need only present `BaseCase()` (Algorithm 39) and `Score()` (Algorithm 40)[18]. The `BaseCase()` function is

[18] In these algorithms, we assume that any point present in a node $\mathcal{N}_i$ will also be present in at least one

Algorithm 39 `BaseCase()` for dual-tree k-means.

Input: query point p_q, reference centroid c_r
Output: distance between p_q and c_r

if $d(p_q, c_r) < \text{ub}(p_q)$ **then**
  $\text{lb}(p_q) \leftarrow \text{ub}(p_q)$
  $\text{ub}(p_q) \leftarrow d(p_q, c_r)$
  $\text{closest}(p_q) \leftarrow c_r$
else if $d(p_q, c_r) < \text{lb}(p_q)$ **then**
  $\text{lb}(p_q) \leftarrow d(p_q, c_r)$
return $d(p_q, c_r)$

Algorithm 40 `Score()` for dual-tree k-means.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: score for node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination can be pruned

{Update the number of pruned nodes, if necessary.}
if $\mathscr{N}_q$ not yet visited and $\mathscr{N}_q$ is not the root node **then**
  $\text{pruned}(\mathscr{N}_q) \leftarrow \text{parent}(\mathscr{N}_q)$
  $\text{lb}(\mathscr{N}_q) \leftarrow \text{lb}(\text{parent}(\mathscr{N}_q))$
if $\text{pruned}(\mathscr{N}_q) = k - 1$ **then return** ∞
$s \leftarrow d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
$c \leftarrow$ any descendant cluster centroid of $\mathscr{N}_r$
if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > \text{ub}(\mathscr{N}_q)$ **then**
  {This cluster node can own no points in this query node.}
  if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) < \text{lb}(\mathscr{N}_q)$ **then**
    {We may improve the lower bound for pruned nodes.}
    $\text{lb}(\mathscr{N}_q) \leftarrow d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
  $\text{pruned}(\mathscr{N}_q) \leftarrow \text{pruned}(\mathscr{N}_q) + |\mathscr{D}_r^p \setminus \{\text{clusters not pruned}\}|$
  $s \leftarrow \infty$
else if $d_{\max}(\mathscr{N}_q, c) < \text{ub}(\mathscr{N}_q)$ **then**
  {We may improve the upper bound.}
  $\text{ub}(\mathscr{N}_q) \leftarrow d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$
  $\text{closest}(\mathscr{N}_q) \leftarrow c$
{Check if all clusters (except one) are pruned.}
if $\text{pruned}(\mathscr{N}_q) = k - 1$ **then return** ∞
return s

straightforward: given a point p_q and a centroid c_r, the distance $d(p_q, c_r)$ is calculated, and $\text{ub}(p_q)$, $\text{lb}(p_q)$, and $\text{closest}(p_q)$ are updated if necessary. `Score()`, however, is significantly

child $\mathscr{N}_c \in \mathscr{C}_i$. It is possible to fully generalize to any tree type, but the exposition is significantly more complex, and our assumption covers most standard tree types anyway.

more complex.

The first stanza (lines 4–6) take the values of pruned(·) and lb(·) from the parent node of $\mathscr{N}_q$; this is necessary because centroids may have been pruned before the node combination $(\mathscr{N}_q, \mathscr{N}_r)$ was visited, and if pruned(·) and lb(·) are not taken from the parent node, then pruned(·) may undercount and lb(·) may be too tight. Next, we prune if the owner of $\mathscr{N}_q$ is already known (line 7). If the minimum distance between any descendant point of $\mathscr{N}_q$ and any descendant centroid of $\mathscr{N}_r$ is greater than the current upper bound for $\mathscr{N}_q$, then we may prune the combination (line 16). In addition, this gives us an opportunity to improve the lower bound (line 14). Note the special handling in line 15: our definition of tree allows points to be held in more than one node; thus, we must avoid double-counting clusters that we prune. The details are left to the implementation[19]. If the node combination cannot be pruned in this way, an attempt is made to update the upper bound (lines 17–20). Instead of using $d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$ (the maximum possible distance between any descendant point of $\mathscr{N}_q$ and any descendant centroid of $\mathscr{N}_r$), we may use a tighter upper bound: select any descendant centroid c from $\mathscr{N}_r$ and use $d_{\max}(\mathscr{N}_q, c)$. This still provides a valid upper bound, and in practice is generally smaller than $d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$. We may simply set closest($\mathscr{N}_q$) to c (line 20): remember that closest($\mathscr{N}_q$) only holds the owner of $\mathscr{N}_q$ if all centroids except one are pruned—in which case the owner *must* be c.

Thus, at the end of the dual-tree algorithm, we know the owner of every node (if it exists) via closest(·) and pruned(·), and we know the owner of every point via closest(·).

7.8.5.2 Updating the centroids with `ExtractCentroids()`

After running the dual-tree algorithm with `BaseCase()` and `Score()`, the centroids must be updated. A simple algorithm to do this is given in Algorithm 41: it is a depth-first recursion through the tree that terminates a branch when a node is owned by a single cluster.

We first initialize the new centroids C to zero; then, starting at the root node, we check

[19]For trees like the *kd*-tree and the metric tree, which do not hold points in more than one node, no special handling is required: we will never prune a cluster twice for a given query node $\mathscr{N}_q$.

Algorithm 41 `UpdateCentroids()` for dual-tree k-means.

1: **Input:** tree $\mathscr{T}$ built on dataset S
2: **Output:** new centroids C
3: $C := \{c_0, \ldots, c_{k-1}\} \leftarrow 0^{k \times d}$
4: $n = 0^k$
5: {A depth-first recursion to calculate centroids. s is a stack.}
6: $s \leftarrow \{\text{root}(\mathscr{T})\}$
7: **while** $|s| > 0$ **do**
8: $\quad \mathscr{N}_i \leftarrow s.\texttt{pop}()$
9: $\quad$ **if** $\text{pruned}(\mathscr{N}_i) = k - 1$ **then**
10: $\quad\quad$ {The node is entirely owned by a cluster.}
11: $\quad\quad j \leftarrow$ index of $\text{closest}(\mathscr{N}_i)$
12: $\quad\quad c_j \leftarrow c_j + |\mathscr{D}_i^p| \, \text{centroid}(\mathscr{N}_i)$
13: $\quad\quad n_j \leftarrow n_j + |\mathscr{D}_i^p|$
14: $\quad$ **else**
15: $\quad\quad$ {The node is not entirely owned by a cluster. Recurse.}
16: $\quad\quad$ **if** $|\mathscr{C}_i| > 0$ **then** $s.\texttt{push}(\mathscr{C}_i)$
17: $\quad\quad$ **else**
18: $\quad\quad\quad$ **for** $p_i \in \mathscr{P}_i$ not yet considered
19: $\quad\quad\quad\quad j \leftarrow$ index of $\text{closest}(p_i)$
20: $\quad\quad\quad\quad c_j \leftarrow c_j + p_i$
21: $\quad\quad\quad\quad n_j \leftarrow n_j + 1$
22: **for** $c_i \in C$, **if** $n_i > 0$ **then** $c_i \leftarrow c_i / n_i$
23: **return** C

if the node is owned by a single cluster. If so (lines 9–13), then the centroid of the node multiplied by the number of descendants is added to the right centroid and the counts for that cluster are updated (lines 12 and 13). Otherwise, we add all the children of the node to the stack, and then add the contributions of each point which has not yet been considered (lines 18–21).

7.8.5.3 Updating the tree with `UpdateTree()`

The next step is updating the bounds in the tree and determining if nodes and points can change owners next iteration; this work is encapsulated in the `UpdateTree()` algorithm. Essentially, this is an implementation of Strategies 3 and 4. Unfortunately, though, this yields a particularly complex recursive algorithm, given in Algorithms 42 and 43 (it is too long for one page). At the end of this algorithm, canchange$(\cdot)$ is set correctly for every

Algorithm 42 `UpdateTree()` for dual-tree k-means.

1: **Input:** $\mathcal{N}_i$, ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, closest$(\cdot)$, canchange$(\cdot)$, centroid movements m
2: **Output:** updated ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, canchange$(\cdot)$

3: canchange$(\mathcal{N}_i) \leftarrow$ `true`
4: **if** $\mathcal{N}_i$ has a parent and canchange(parent$(\mathcal{N}_i)$) = `false` **then**
5: {Use the parent's bounds.}
6: closest$(\mathcal{N}_i) \leftarrow$ closest(parent$(\mathcal{N}_i)$)
7: $j \leftarrow$ index of closest$(\mathcal{N}_i)$
8: ub$(\mathcal{N}_i) \leftarrow$ ub$(\mathcal{N}_i) + m_j$
9: lb$(\mathcal{N}_i) \leftarrow$ lb$(\mathcal{N}_i) + \max_i m_i$
10: canchange$(\mathcal{N}_i) \leftarrow$ `false`

11: **else if** pruned$(\mathcal{N}_i) = k - 1$ **then**
12: {$\mathcal{N}_i$ is owned by a single cluster. Can that owner change next iteration?}
13: $j \leftarrow$ index of closest$(\mathcal{N}_i)$
14: ub$(\mathcal{N}_i) \leftarrow$ ub$(\mathcal{N}_i) + m_j$
15: lb$(\mathcal{N}_i) \leftarrow \max\left(\text{lb}(\mathcal{N}_i) - \max_i m_i, \min_{k \neq j} d(c_k, c_j)/2\right)$
16: **if** ub$(\mathcal{N}_i) <$ lb$(\mathcal{N}_i)$ **then**
17: {The owner cannot change next iteration.}
18: canchange$(\mathcal{N}_i) \leftarrow$ `false`
19: **else**
20: {Tighten the upper bound and try to prune again.}
21: ub$(\mathcal{N}_i) \leftarrow \min\left(\text{ub}(\mathcal{N}_i), d_{\max}(\mathcal{N}_i, c_j)\right)$
22: **if** ub$(\mathcal{N}_i) <$ lb$(\mathcal{N}_i)$ **then** canchange$(\mathcal{N}_i) \leftarrow$ `false`

23: **else**
24: $j \leftarrow$ index of closest$(\mathcal{N}_i)$
25: ub$(\mathcal{N}_i) \leftarrow$ ub$(\mathcal{N}_i) + m_j$
26: lb$(\mathcal{N}_i) \leftarrow$ lb$(\mathcal{N}_i) - \max_k m_k$

27: {Recurse into each child.}
28: **for each** child $\mathcal{N}_c$ of $\mathcal{N}_i$**, call** `UpdateTree(`$\mathcal{N}_c$`)`

29: {The function is too long for one page...}
30: **call** `UpdateTreePartTwo(`$\mathcal{N}_i$`)`

point and node.

The first if statement (lines 4–10) catches the case where the parent cannot change owner next iteration; in this case, the parent's upper bound and lower bound can be taken as valid bounds. In addition, the upper and lower bounds are adjusted to account for cluster movement between iterations, so that the bounds are valid for next iteration.

Algorithm 43 `UpdateTreePartTwo()` for dual-tree k-means.

1: **Input:** $\mathcal{N}_i$, ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, closest$(\cdot)$, canchange$(\cdot)$, centroid movements m
2: **Output:** updated ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, canchange$(\cdot)$
3: {Try to determine points whose owner cannot change if $\mathcal{N}_i$ can change owners.}
4: **if** canchange$(\mathcal{N}_i)$ = `true` **then**
5: **for** $p_i \in \mathcal{P}_i$ **do**
6: $j \leftarrow$ index of closest(p_i)
7: ub$(p_i) \leftarrow$ ub$(p_i) + m_j$
8: lb$(p_i) \leftarrow \min\left(\text{lb}(p_i) - \max_k m_k, \min_{k \neq j} d(c_k, c_j)/2\right)$
9: **if** ub$(p_i) <$ lb(p_i) **then**
10: canchange$(p_i) \leftarrow$ `false`
11: **else**
12: {Tighten the upper bound and try again.}
13: ub$(p_i) \leftarrow \min\left(\text{ub}(p_i), d(p_i, c_j)\right)$
14: **if** ub$(p_i) <$ lb(p_i) **then**
15: canchange$(p_i) \leftarrow$ `false`
16: **else**
17: {Point cannot be pruned.}
18: ub$(p_i) \leftarrow \infty$
19: lb$(p_i) \leftarrow \infty$
20: **else**
21: **for** $p_i \in \mathcal{P}_i$ where canchange(p_i) = `false` **do**
22: {Maintain upper and lower bounds for points whose owner cannot change.}
23: $j \leftarrow$ index of closest(p_i)
24: ub$(p_i) \leftarrow$ ub$(p_i) + m_j$
25: lb$(p_i) \leftarrow$ lb$(p_i) - \max_k m_k$
26: **if** canchange$(\cdot)$ = `false` for all children $\mathcal{N}_c$ of $\mathcal{N}_i$ and all points $p_i \in \mathcal{P}_i$ **then**
27: canchange$(\mathcal{N}_i) \leftarrow$ `false`
28: **if** canchange$(\mathcal{N}_i)$ = `true` **then**
29: pruned$(\mathcal{N}_i) \leftarrow 0$

If the node $\mathcal{N}_i$ has an owner, the algorithm then attempts to use the pruning rules established in Equations 176 and 178 to determine if the owner of $\mathcal{N}_i$ can change next iteration. If not, canchange$(\mathcal{N}_i)$ is set to `false` (line 18). On the other hand, if the pruning check fails, the upper bound is tightened and the pruning check is performed a second time. It is worth noting that $d_{\max}(\mathcal{N}_i, c_j)$ may not actually be less than the current value of ub$(\mathcal{N}_i)$, which is why the min is necessary.

After recursing into the children of $\mathcal{N}_i$, if $\mathcal{N}_i$ could have an owner change, each point

is individually checked using the same approach (lines 4–20). However, there is a slight difference: if a point's owner can change, the upper and lower bounds must be set to ∞ (lines 18–19). This is only necessary with points; `BaseCase()` does not take bounding information from previous iterations into account, because no work can be avoided in that way.

Then, we may set canchange($\mathscr{N}_i$) to `false` if every point in $\mathscr{N}_i$ and every child of $\mathscr{N}_i$ cannot change owners (and the points and nodes do not necessarily have to have the same owner). Otherwise, we must set pruned($\mathscr{N}_i$) to 0 for the next iteration.

7.8.5.4 Coalescing (and decoalescing) the tree

Once `UpdateTree()` sets the correct value of canchange($\cdot$) for every point and node, we may coalesce the tree for the next iteration with the `CoalesceTree()` function. Coalescing the tree is straightforward: essentially, we simply remove any nodes from the tree where canchange($\cdot$) is `false`. This leaves us with a smaller tree that has no nodes where canchange($\cdot$) is `false`.

This can be accomplished via a single pass over the tree. A simple implementation is given in Algorithm 44. `DecoalesceTree()` may be implemented by simply restoring a pristine copy of the tree which was cached right before `CoalesceTree()` is called.

7.8.6 Theoretical results

In this subsection, we show theoretical results for the dual-tree k-means algorithm, including a correctness proof, bounds on per-iteration runtime, and memory bounds. We will start with the (quite complex) correctness proof.

7.8.6.1 Correctness of the algorithm

We will individually prove the correctness of various pieces of the dual-tree k-means algorithm, and then we will prove the main correctness result. Correctness is proven not just for a certain type of tree but for all types of trees that satisfy the definition of *space tree* and all types of traversals that satisfy the definition of *pruning dual-tree traversal* as given

Algorithm 44 `CoalesceTree()` for dual-tree k-means.

Input: tree $\mathscr{T}$
Output: coalesced tree $\mathscr{T}$
{A depth-first recursion to hide nodes where canchange$(\cdot)$ is `false`.}
$s \leftarrow \{\text{root}(\mathscr{T})\}$
while $|s| > 0$ **do**
  $\mathscr{N}_i \leftarrow s$.pop()
  {Special handling is required for leaf nodes and the root node.}
  if $|\mathscr{C}_i| = 0$ **then**
    continue
  else if $\mathscr{N}_i$ is the root node **then**
    for $\mathscr{N}_c \in \mathscr{C}_i$ **do**
      s.push$(\mathscr{N}_c)$
  {See if children can be removed.}
  for $\mathscr{N}_c \in \mathscr{C}_i$ **do**
    if canchange$(\mathscr{N}_c)$ = `false` **then**
      remove child $\mathscr{N}_c$
    else
      s.push$(\mathscr{N}_c)$
  {If only one child is left, then this node is unnecessary.}
  if $|\mathscr{C}_i| = 1$ **then**
    add child to parent$(\mathscr{N}_i)$
    remove $\mathscr{N}_i$ from parent$(\mathscr{N}_i)$'s children
return $\mathscr{T}$

in Chapter 3.

Lemma 7. *A pruning dual-tree traversal which uses* `BaseCase()` *as given in Algorithm 39 and* `Score()` *as given in Algorithm 40 which starts with valid* ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, *and* closest$(\cdot)$ *for each node* $\mathscr{N}_i \in \mathscr{T}$, *and* $\text{ub}(p_q) = \text{lb}(p_q) = \infty$ *for each point* $p_q \in S$, *will satisfy the following conditions upon completion:*

- *For every* $p_q \in S$ *that is a descendant of a node* $\mathscr{N}_i$ *that has been pruned (that is,* $\text{pruned}(\mathscr{N}_i) = k - 1$*),* $\text{ub}(\mathscr{N}_i)$ *is an upper bound on the distance between* p_q *and its closest centroid, and* $\text{closest}(\mathscr{N}_i)$ *is the owner of* p_q.
- *For every* $p_q \in S$ *that is not a descendant of any node that has been pruned,* $\text{ub}(p_q)$ *is an upper bound on the distance between* p_q *and its closest centroid, and* $\text{closest}(p_q)$

is the owner of p_q.

- *For every $p_q \in S$ that is a descendant of a node $\mathscr{N}_i$ that has been pruned (that is,* $\text{pruned}(\mathscr{N}_i) = k - 1$*), $\text{lb}(\mathscr{N}_i)$ is a lower bound on the distance between p_q and its second closest centroid.*

- *For every $p_q \in S$ that is not a descendant of any node that has been pruned, the quantity* $\min(\text{lb}(p_q), \text{lb}(\mathscr{N}_q))$ *where $\mathscr{N}_q$ is a node such that $p_q \in \mathscr{P}_q$ is a lower bound on the distance between p_q and its second closest centroid.*

Proof. It is easiest to consider each condition individually. Thus, we will first consider the upper bound on the distance to the closest cluster centroid. Consider some p_q and suppose that the closest cluster centroid to p_q is c^*.

Now, suppose first that the point p_q is a descendant point of a node $\mathscr{N}_q$ that has been pruned. We must show, then, that c^* is closest($\mathscr{N}_q$). Take $R = \{\mathscr{N}_{r0}, \mathscr{N}_{r1}, \ldots, \mathscr{N}_{rj}\}$ to be the set of reference nodes visited during the traversal with $\mathscr{N}_q$ as a query node; that is, the combinations $(\mathscr{N}_q, \mathscr{N}_{ri})$ were visited for all $\mathscr{N}_{ri} \in R$. Any $\mathscr{N}_{ri}$ is pruned only if

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_{ri}) > \text{ub}(\mathscr{N}_i) \tag{183}$$

according to line 10 of `Score()`. Thus, as long as $\text{ub}(\mathscr{N}_i)$ is a valid upper bound on the closest cluster distance for every descendant point in $\mathscr{N}_q$, then no nodes are incorrectly pruned. It is easy to see that the upper bound is valid: initially, it is valid by assumption; each time the bound is updated with some node $\mathscr{N}_{ri}$ (on lines 19 and 20), it is set to $d_{\max}(\mathscr{N}_i, c)$ where c is some descendant centroid of $\mathscr{N}_{ri}$. This is clearly a valid upper bound, since c cannot be any closer to any descendant point of $\mathscr{N}_i$ than c^*. We may thus conclude that no node is incorrectly pruned from R; we may apply this reasoning recursively to the $\mathscr{N}_q$'s ancestors to see that no reference node is incorrectly pruned.

When a node is pruned from R, the number of pruned clusters for $\mathscr{N}_q$ is updated: the count of all clusters not previously pruned by $\mathscr{N}_q$ (or its ancestors) is added. We cannot

double-count the pruning of a cluster; thus the only way that pruned($\mathscr{N}_q$) can be equal to $k-1$ is if every centroid except one is pruned. The centroid which is not pruned will be the nearest centroid c^*, regardless of if closest($\mathscr{N}_q$) was set during this traversal or still holds its initial value, and therefore it must be true that ub($\mathscr{N}_q$) is an upper bound on the distance between p_q and c^*, and closest($\mathscr{N}_q$) = c^*.

This allows us to finally conclude that if p_q is a descendant of a node $\mathscr{N}_q$ that has been pruned, then ub($\mathscr{N}_q$) contains a valid upper bound on the distance between p_q and its closest cluster centroid, and closest($\mathscr{N}_q$) is that closest cluster centroid.

Now, consider the other case, where p_q is not a descendant of any node that has been pruned. Take $\mathscr{N}_i$ to be any node containing p_q[20]. We have already reasoned that any cluster centroid node that could possibly contain the closest cluster centroid to p_q cannot have been pruned; therefore, by the definition of pruning dual-tree traversal, we are guaranteed that `BaseCase()` will be called with p_q as the query point and the closest cluster centroid as the reference point. This will then cause ub(p_q) to hold the distance to the closest cluster centroid—assuming ub(p_q) is always valid, which it is even at the beginning of the traversal because it is initialized to ∞—and closest(p_q) to hold the closest cluster centroid.

Therefore, the first two conditions are proven. The third and fourth conditions, for the lower bounds, require a slightly different strategy.

There are two ways lb($\mathscr{N}_q$) is modified: first, at line 14, when a node combination is pruned, and second, at line 6 when the lower bound is taken from the parent. Again, consider the set $R = \{\mathscr{N}_{r0}, \mathscr{N}_{r1}, \ldots, \mathscr{N}_{rj}\}$ which is the set of reference nodes visited during the traversal with $\mathscr{N}_q$ as a query node. Call the set of reference nodes that were pruned R^p. At the end of the traversal, then,

[20]Note that the meaning here is not that p_q is a descendant of $\mathscr{N}_i$ ($p_i \in \mathscr{D}_i^p$), but instead that p_q is held directly in $\mathscr{N}_i$: $p_q \in \mathscr{P}_i$.

$$\text{lb}(\mathscr{N}_q) \leq \min_{\mathscr{N}_{ri} \in R^p} d_{\min}(\mathscr{N}_q, \mathscr{N}_{ri}) \tag{184}$$

$$\leq \min_{c_k \in C^p} d_{\min}(\mathscr{N}_q, c_k) \tag{185}$$

where C^p is the set of centroids that are descendants of nodes in R^p. Applying this reasoning recursively to the ancestors of $\mathscr{N}_q$ shows that at the end of the dual-tree traversal, $\text{lb}(\mathscr{N}_q)$ will contain a lower bound on the distance between any descendant point of $\mathscr{N}_q$ and any pruned centroid. Thus, if $\text{pruned}(\mathscr{N}_q) = k - 1$, then $\text{lb}(\mathscr{N}_q)$ will contain a lower bound on the distance between any descendant point in $\mathscr{N}_q$ and its second closest centroid. So if we consider some point p_q which is a descendant of $\mathscr{N}_q$ and $\mathscr{N}_q$ is pruned ($\text{pruned}(\mathscr{N}_q) = k-1$), then $\text{lb}(\mathscr{N}_q)$ is indeed a lower bound on the distance between p_q and its second closest centroid.

Now, consider the case where p_q is not a descendant of any node that has been pruned, and take $\mathscr{N}_q$ to be some node that owns p_q (that is, $p_q \in \mathscr{P}_q$). In this case, `BaseCase()` will be called with every centroid that has not been pruned. So $\text{lb}(\mathscr{N}_q)$ is a lower bound on the distance between p_q and every pruned centroid, and $\text{lb}(p_q)$ will be a lower bound on the distance between p_q and the second-closest non-pruned centroid, due to the structure of the `BaseCase()` function. Therefore, $\min(\text{lb}(p_q), \text{lb}(\mathscr{N}_q))$ must be a lower bound on the distance between p_q and its second closest centroid.

Finally, we may conclude that each item in the theorem holds. □

Next, we must prove that `UpdateTree()` functions correctly.

Lemma 8. *In the context of Algorithm 38, given a tree $\mathscr{T}$ with all associated bounds* $\text{ub}(\cdot)$ *and* $\text{lb}(\cdot)$ *and information* $\text{pruned}(\cdot)$, $\text{closest}(\cdot)$, *and* $\text{canchange}(\cdot)$, *a run of* `UpdateTree()` *as given in Algorithm 42 will have the following effects:*

- *For every node $\mathscr{N}_i$,* $\text{ub}(\mathscr{N}_i)$ *will be a valid upper bound on the distance between any descendant point of $\mathscr{N}_i$ and its nearest centroid next iteration.*

- *For every node* $\mathcal{N}_i$, $\text{lb}(\mathcal{N}_i)$ *will be a valid lower bound on the distance between any descendant point of* $\mathcal{N}_i$ *and any pruned centroid next iteration.*

- *A node* $\mathcal{N}_i$ *will only have* $\text{canchange}(\mathcal{N}_i) = \texttt{false}$ *if the owner of any descendant point of* $\mathcal{N}_i$ *cannot change next iteration.*

- *A point* p_i *will only have* $\text{canchange}(p_i) = \texttt{false}$ *if the owner of* p_i *cannot change next iteration.*

- *Any point* p_i *with* $\text{canchange}(p_i) = \texttt{true}$ *that does not belong to any node* $\mathcal{N}_i$ *with* $\text{canchange}(\mathcal{N}_i) = \texttt{false}$ *will have* $\text{ub}(p_i) = \text{lb}(p_i) = \infty$*, as required by the dual-tree traversal.*

- *Any node* $\mathcal{N}_i$ *with* $\text{canchange}(\mathcal{N}_i) = \texttt{false}$ *at the end of* `UpdateTree()` *will have* $\text{pruned}(\mathcal{N}_i) = 0$.

Proof. Each point is best considered individually. It is important to remember during this proof that the centroids have been updated, but the bounds have not. So any cluster centroid c_i is already set for next iteration. Take c_i^l to mean the cluster centroid c_i *before* adjustment (that is, the old centroid). Also take $\text{ub}^l(\cdot)$, $\text{lb}^l(\cdot)$, $\text{pruned}^l(\cdot)$, and $\text{canchange}^l(\cdot)$ to be the values at the time `UpdateTree()` is called, before any of those values are changed. Due to the assumptions in the statement of the lemma, each of these quantities is valid.

Suppose that for some node $\mathcal{N}_i$, $\text{closest}(\mathcal{N}_i)$ is some cluster c_j. For $\text{ub}(\mathcal{N}_i)$ to be valid for next iteration, we must guarantee that $\text{ub}(\mathcal{N}_i) \geq \max_{p_q \in \mathcal{D}_q^p} d(p_q, c_j)$ at the end of `UpdateTree()`. There are four ways $\text{ub}(\mathcal{N}_i)$ is updated: it may be taken from the parent and adjusted (line 8), it may be adjusted before a prune attempt (line 14), it may be tightened after a failed prune attempt (line 21), or it may be adjusted without a prune attempt (line 25). If we can show that each of these four ways always results in $\text{ub}(\mathcal{N}_i)$ being valid, then the first condition of the theorem holds.

If $\text{ub}(\mathscr{N}_i)$ is adjusted in line 14 or 25, the resulting value of $\text{ub}(\mathscr{N}_i)$, assuming that $\text{closest}(\mathscr{N}_i) = c_j$, is

$$\begin{aligned}
\text{ub}(\mathscr{N}_i) &= \text{ub}^l(\mathscr{N}_i) + m_j && (186)\\
&\geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j^l) + m_j && (187)\\
&\geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j) && (188)
\end{aligned}$$

where the last step follows by the triangle inequality: $d(c_j, c_j^l) = m_j$. Therefore those two updates to $\text{ub}(\mathscr{N}_i)$ result in valid upper bounds for next iteration. If $\text{ub}(\mathscr{N}_i)$ is recalculated, in line 21, then we are guaranteed that $\text{ub}(\mathscr{N}_i)$ is valid because

$$d_{\max}(\mathscr{N}_i, c_j) \geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j). \quad (189)$$

We may therefore conclude that $\text{ub}(\mathscr{N}_i)$ is correct for the root of the tree, because line 8 can never be reached. Reasoning recursively, we can see that any upper bound passed from the parent must be valid. Therefore, the first item of the lemma holds.

Next, we will consider the lower bound, using a similar strategy. We must show that

$$\text{lb}(\mathscr{N}_i) \leq \min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p) \quad (190)$$

where C_p is the set of centroids pruned by $\mathscr{N}_i$ and ancestors during the last dual-tree traversal. The lower bound can be taken from the parent in line 10 and adjusted, it can be adjusted before a prune attempt in line 15 or in a similar way without a prune attempt in line 26. The last adjustment can easily be shown to be valid:

$$\begin{aligned}
\text{lb}(\mathscr{N}_i) &= \text{lb}^l(\mathscr{N}_i) - \max_k m_k && (191)\\
&\leq \left(\min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p^l) \right) - \max_k m_k && (192)\\
&\leq \min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p) && (193)
\end{aligned}$$

which follows by the triangle inequality: $d(c_p^l, c_p) \leq \max_k m_k$. Line 15 is slightly more complex; we must also consider the term $\min_{k \neq j} d(c_k, c_j)/2$. Suppose that

$$\min_{k \neq j} d(c_k, c_j)/2 > \text{lb}^l(\mathcal{N}_i) + \max_k m_k. \tag{194}$$

We may use the triangle inequality ($d(p_q, c_k) \leq d(c_j, c_k) + d(p_q, c_j)$) to show that if this is true, the second closest centroid c_k is such that $d(p_q, c_k) > 2d(c_k, c_j)$ and therefore $\min_{k \neq j} d(c_k, c_j)/2$ is also a valid lower bound. We can lastly use the same recursive argument from the upper bound case to show that the second item of the lemma holds.

Showing the correctness of canchange($\mathcal{N}_i$) is straightforward: we know that ub($\mathcal{N}_i$) and lb($\mathcal{N}_i$) are valid for next iteration by the time any checks to set canchange($\mathcal{N}_i$) to `false` happens, due to the discussion above. The situations where canchange($\mathcal{N}_i$) is set to `false`, in line 16 and 22, are simply applications of Equations 176 and 178, and are therefore valid. There are two other ways canchange($\mathcal{N}_i$) can be set to `false`. The first is on line 10, and this is easily shown to be valid: if a parent's owner cannot change, then a child's owner cannot change either. The other way to set canchange($\mathcal{N}_i$) to `false` is in line 27. This is only possible if all points in $\mathcal{P}_i$ and all children of $\mathcal{N}_i$ have canchange($\cdot$) set to `false`; thus, no descendant point of $\mathcal{N}_i$ can change owner next iteration, and we may set canchange($\mathcal{N}_i$) to `false`.

Next, we must show that canchange(p_i) = `false` only if the owner of p_i cannot change next iteration. If canchange$^l(p_i)$ = `true`, then due to Lemma 7, ub$^l(p_i)$ and lb$^l(p_i)$ will be valid bounds. In this case, we may use similar reasoning to show that ub(p_i) and lb(p_i) are valid, and then we may see that the pruning attempts at line 9 and 14 are valid. Now, consider the other case, where canchange$^l(p_i)$ = `false`. Then, ub$^l(p_i)$ and lb$^l(p_i)$ will not have been modified by the dual-tree traversal, and will hold the values set in the previous run of `UpdateTree()`. As long as those values are valid, then the fourth item holds.

The checks to see if canchange(p_i) can be set to `false` (from lines 4 to 20) are only reached if canchange($\mathcal{N}_i$) is `true`. We already have shown that ub(p_i) and lb(p_i) are set

correctly in that stanza. The other case is if canchange($\mathscr{N}_i$) is `false`. In this case, lines 21 to 25. It is easy to see using similar reasoning to all previous cases that these lines result in valid ub(p_i) and lb(p_i). Therefore, the fourth item does hold.

The fifth item is taken care of in line 18 and 19. Given some point p_i with canchange(p_i) set to`true`, and where p_i does not belong to any node $\mathscr{N}_i$ where canchange($\mathscr{N}_i$) = `false`, these two lines must be reached, and therefore the fifth item holds.

The last item holds trivially—any node $\mathscr{N}_i$ where canchange($\mathscr{N}_i$) = `true` will have pruned($\mathscr{N}_i$) set to 0 on line 29. □

Showing that the three auxiliary methods `CoalesceTree()`, `DecoalesceTree()`, and `UpdateCentroids()` function correctly follows directly from the algorithm descriptions. Therefore, we are ready to show the main correctness result.

Theorem 18. *A single iteration of dual-tree k-means as given in Algorithm 38 will produce exactly the same results as the standard brute-force $O(kN)$ implementation.*

Proof. We may use the previous lemmas to flesh out our earlier proof sketch.

First, we know that the dual-tree algorithm (line 9) produces correct results for ub(·), lb(·), pruned(·), and closest(·) for every point and node, due to Lemma 7. Next, we know that `UpdateTree()` maintains the correctness of those four quantities and only marks canchange(·) to `false` when the node or point truly cannot change owner, due to Lemma 8. Next, we know from earlier discussion that `CoalesceTree()` and `DecoalesceTree()` do not affect the results of the dual-tree algorithm because the only nodes and points removed are those where canchange(·) = `false`. We also know that `UpdateCentroids()` produces centroids correctly. Therefore, the results from Algorithm 38 are identical to those of a brute-force $O(kN)$ k-means implementation. □

7.8.6.2 Per-iteration runtime bound

Next, we consider the runtime of the algorithm, using adaptive algorithm analysis techniques in order to bound the per-iteration running time of Algorithm 38. In order to use

these techniques, this runtime bound assumes the use of the cover tree and the standard pruning cover-tree traversal (see Algorithm 8).

These bounds are based on techniques and quantities introduced comprehensively in Section 5.2 and in other works [57, 135, 133]; the results are with respect to the expansion constant c_k of the centroids, which is a measure of intrinsic dimension. c_{qk} is a related quantity: the largest expansion constant of C plus any point in the dataset. Our results also depend on the imbalance of the tree $i_t(\mathscr{T})$, which in practice generally scales linearly in N [135].

We may first use the runtime bound result from nearest neighbor search (Section 7.1.4) to bound the running time of the dual-tree algorithm part of dual-tree k-means.

Theorem 19. *The dual-tree k-means algorithm with* `BaseCase()` *as in Algorithm 39 and* `Score()` *as in Algorithm 40, with a point set S_q that has expansion constant c_q and size N, and k centroids C with expansion constant c_k, takes no more than $O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T}_q)))$ time.*

Proof. Both `Score()` and `BaseCase()` for dual-tree k-means can be performed in $O(1)$ time. In addition, the pruning of `Score()` for dual-tree k-means is at least as tight as `Score()` for nearest neighbor search: the pruning rule in Equation 176 is equivalent to the pruning rule for nearest neighbor search. Therefore, dual-tree k-means can visit no more nodes than nearest neighbor search would with query set S_q and reference set C. Lastly, note that the range of pairwise distances of C will be entirely contained in the range of pairwise distances in S_q, to see that we can use the result of Theorem 4. Adapting that result, then, yields the statement of the algorithm. □

The expansion constant of the centroids, c_k, can be expected to behave similarly to the expansion constant c_q of the dataset, because the centroids will arise from a similar distribution as the points. It is thus reasonable to assume c_k does not scale with k, if it is already assumed that c_q does not scale with N. It is also reasonable to assume c_{qk} does not

scale with N or k, for the same reasons.

Next, we turn to bounding the entire algorithm.

Theorem 20. *A single iteration of the dual-tree k-means algorithm on a dataset S_q using the cover tree $\mathscr{T}$, the standard cover tree pruning dual-tree traversal,* `BaseCase()` *as given in Algorithm 39,* `Score()` *as given in Algorithm 40, will take no more than*

$$O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})) + c_k^9 k \log k) \tag{195}$$

time, where c_k is the expansion constant of the centroids, c_{qk} is defined as in Theorem 19, and $i_t(\mathscr{T})$ is the imbalance of the tree as defined in Definition 10.

Proof. Consider each of the steps of the algorithm individually:

- `CoalesceNodes()` can be performed in a single pass of the cover tree $\mathscr{N}$, which takes $O(N)$ time.
- Building a tree on the centroids ($\mathscr{T}_c$) takes $O(c_k^6 k \log k)$ time due to the result for cover tree construction time [57].
- The dual-tree algorithm takes $O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})))$ time due to Theorem 19.
- `DecoalesceNodes()` can be performed in a single pass of the cover tree $\mathscr{N}$, which takes $O(N)$ time.
- `UpdateCentroids()` can be performed in a single pass of the cover tree $\mathscr{N}$, so it also takes $O(N)$ time.
- `UpdateTree()` depends on the calculation of how much each centroid has moved; this costs $O(k)$ time. In addition, we must find the nearest centroid of every centroid; this is nearest neighbor search, and we may use the runtime bound for monochromatic nearest neighbor search for cover trees from [133], so this costs $O(c_k^9 k)$ time. Lastly, the actual tree update visits each node once and iterates over each point in

the node. Cover tree nodes only hold one point, so each visit costs $O(1)$ time, and with $O(N)$ nodes, the entire update process costs $O(N)$ time. When we consider the preprocessing cost too, the total cost of `UpdateTree()` per iteration is $O(c_k^9 k + N)$.

We may combine these into a final result:

$$O(N) + O(c_k^6 k \log k) + O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T}))) + O(N) + O(N) + O(c_k^9 k + N) \tag{196}$$

and after simplification, we get the statement of the theorem:

$$O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})) + c_k^9 k \log k). \tag{197}$$

□

Therefore, we see that under some assumptions on the data, we can bound the runtime of the dual-tree k-means algorithm to something tighter than $O(kN)$ per iteration. As expected, we are able to amortize the cost of k across all N nodes, giving amortized $O(1)$ search for the nearest centroid per point in the dataset. This is similar to the results for nearest neighbor search, which obtain amortized $O(1)$ search for a single query point. Also similar to the results for nearest neighbor search is that the search time may, in the worst case, degenerate to $O(kN + k^2)$ when the assumptions on the dataset are not satisfied. However, empirical results [67, 61, 68, 57] show that well-behaved datasets are common in the real world, and thus degeneracy of the search time is uncommon.

Comparing this bound with the bounds for other k-means algorithms is somewhat difficult; first, none of the other algorithms have bounds which are adaptive to the characteristics of the dataset. It is possible that the blacklist algorithm could be refactored to use the cover tree, but even if that was done it is not completely clear how the running time could be bounded. How to apply the expansion constant to an analysis of Hamerly's algorithm and Elkan's algorithm is also unclear at the time of this writing.

Lastly, the bound we have shown above is potentially loose. We have reduced dual-tree k-means to the problem of nearest neighbor search, but our pruning rules are tighter. Dual-tree nearest neighbor search assumes that every query node will be visited (this is where the $O(N)$ in the bound comes from), but dual-tree k-means can prune a query node entirely if all but one cluster is pruned (Strategy 2). These bounds do not take this pruning strategy into account, and they also do not consider the fact that coalescing the tree can greatly reduce its size. These would be interesting directions for future theoretical work.

7.8.6.3 Memory usage bounds

Bounding the memory usage of dual-tree k-means is comparatively a walk in the park.

Theorem 21. *Algorithm 38 uses no more than $O(N+k)$ memory when cover trees are used.*

Proof. This proof is straightforward. Because a cover tree on N points takes $O(N)$ space, holding the tree built on the points and all associated bounds takes $O(N)$ space. Holding the tree built on the centroids takes $O(k)$ space. The dataset takes $O(N)$ space, and the centroids take $O(k)$ space. Therefore, the theorem holds. □

7.8.7 Experiments

The next thing to consider is the empirical performance of the algorithm. The `kmeans` program in **mlpack** [87] implements each of the k-means algorithms we have discussed to this point. In our experiments, we run it as follows:

```
$ kmeans -i dataset.csv -I centroids.csv -c $k -v -e -a $algorithm
```

where `$k` is the number of clusters and `$algorithm` is one of a handful of choices:

- `elkan`: Elkan's algorithm [220],
- `hamerly`: Hamerly's algorithm [221],
- `blacklist`: Pelleg and Moore's blacklist algorithm [39],

Table 25: Dataset information for dual-tree k-means.

			tree build time	
Dataset	N	d	*kd*-tree	cover tree
cloud	2048	10	0.001s	0.005s
cup98b	95413	56	1.640s	32.41s
birch3	100000	2	0.037s	2.125s
phy	150000	78	4.138s	22.99s
power	2075259	7	7.342s	1388s
lcdm	6000000	3	4.345s	6214s

- `dualtree-kd`: the dual-tree algorithm using *kd*-trees, and
- `dualtree-ct`: the dual-tree algorithm using cover trees.

Each algorithm is implemented in C++ in the same framework, so no algorithm has an implementation quality advantage. We use a variety of k values on mostly real-world datasets; details are shown in Table 25. These datasets are each from the UCI dataset repository [134], with the exception of the (synthetic) birch3 dataset [145] and the LCDM dataset [146]. Table 25 also contains the time taken to build a *kd*-tree (for `blacklist` and `dualtree-kd`) and a cover tree (for `dualtree-ct`). Cover tree construction is significantly more complex than *kd*-tree construction[21]; this accounts for the long cover tree build time. Even so, the tree only needs to be built once during the k-means run. If results are required for multiple values of k—such as in the X-means algorithm [229]—then the tree built on the points may be re-used.

Average runtime per iteration results are shown in Table 26. The amount of work that is being pruned away is somewhat unclear from the runtime results, because the `elkan` and `hamerly` algorithms access points linearly and thus benefit from cache effects; this is not true of the tree-based algorithms. Therefore, the average number of distance calculations per iteration are also included in the results.

[21] Izbicki and Shelton recently proposed a parallel cover tree construction algorithm which might be useful in this situation [228]. They also provide a few improvements to the cover tree construction algorithm which can make search faster for nearest neighbor search; whether or not those improvements would be useful here is as of yet unknown, but in my personal opinion it's at least worth trying someday.

Table 26: Empirical results for k-means.

			avg. per-iteration runtime (distance calculations)				
dataset	k	**iter.**	`elkan`	`hamerly`	`blacklist`	`dualtree-kd`	`dualtree-ct`
cloud	3	8	1.50e-4s (867)	1.11e-4s (1.01k)	**4.68e-5s** (302)	1.27e-4s (**278**)	2.77e-4s (443)
cloud	10	14	2.09e-4s (**1.52k**)	1.92e-4s (4.32k)	**1.55e-4s** (2.02k)	3.69e-4s (1.72k)	5.36e-4s (2.90k)
cloud	50	19	5.87e-4s (**2.57k**)	**5.30e-4s** (21.8k)	8.20e-4s (12.6k)	1.23e-3s (5.02k)	1.09e-3s (9.84k)
cup98b	50	224	0.0445s (**25.9k**)	0.0557s (962k)	**0.0409s** (277k)	0.0955s (254k)	0.1089s (436k)
cup98b	250	168	**0.1972s** (**96.8k**)	0.4448s (8.40M)	0.2033s (1.36M)	0.4585s (1.38M)	0.3237s (2.73M)
cup98b	750	116	1.1719s (**373k**)	1.8778s (36.2M)	**0.6365s** (4.11M)	1.2847s (4.16M)	0.8056s (81.4M)
birch3	50	129	0.0194s (**24.2k**)	0.0093s (566k)	**0.0030s** (42.7k)	0.0082s (37.4k)	0.0378s (67.9k)
birch3	250	812	0.0895s (**42.8k**)	0.0314s (2.59M)	**0.0164s** (165k)	0.0183s (79.7k)	0.0485s (140k)
birch3	750	373	0.3253s (292k)	0.0972s (8.58M)	0.0554s (450k)	**0.02989s** (**126k**)	0.0581s (235k)
phy	50	34	0.0668s (82.3k)	0.1064s (1.38M)	**0.0081s** (**33.0k**)	0.02689s (67.8k)	0.0945s (188k)
phy	250	38	0.1627s (121k)	0.4634s (6.83M)	**0.0249s** (104k)	0.0398s (**90.4k**)	0.1023s (168k)
phy	750	35	0.7760s (**410k**)	2.9192s (43.8M)	**0.2478s** (1.19M)	0.2939s (1.10M)	0.3330s (1.84M)
covertype	50	405	0.2660s (**180k**)	0.1970s (3.13M)	**0.1220s** (747k)	0.1951s (419k)	0.4252s (656k)
covertype	100	455	0.4625s (**224k**)	0.5347s (9.71M)	**0.2025s** (1.15M)	0.3152s (754k)	0.5523s (1.31M)
covertype	500	1000	2.0546s (**295k**)	3.4966s (69.0M)	**0.7583s** (3.49M)	0.8989s (2.12M)	0.8890s (4.12M)
power	25	4	0.3872s (2.98M)	0.2880s (12.9M)	**0.0301s** (216k)	0.0950s (**87.4k**)	0.6658s (179k)
power	250	101	2.6532s (425k)	0.1868s (7.83M)	0.1504s (1.13M)	**0.1354s** (**192k**)	0.6405s (263k)
power	1000	870	*out of memory*	6.2407s (389M)	0.6657s (2.98M)	**0.4115s** (**1.57M**)	1.1799s (4.81M)
power	5000	504	*out of memory*	29.816s (1.87B)	4.1597s (11.7M)	**1.0580s** (**3.85M**)	1.7070s (12.3M)
power	15000	301	*out of memory*	111.74s (6.99B)	*out of memory*	**2.3708s** (**8.65M**)	2.9472s (30.9M)
lcdm	500	507	*out of memory*	6.4084s (536M)	0.9347s (4.20M)	**0.7574s** (**3.68M**)	2.9428s (7.03M)
lcdm	1000	537	*out of memory*	16.071s (1.31B)	2.0345s (5.93M)	**0.9827s** (**5.11M**)	3.3482s (10.0M)
lcdm	5000	218	*out of memory*	64.895s (5.38B)	12.909s (16.2M)	**1.8972s** (**8.54M**)	3.9110s (19.0M)
lcdm	20000	108	*out of memory*	298.55s (24.7B)	*out of memory*	**4.1911s** (**17.8M**)	5.5771s (43.2M)

It is immediately clear that for large datasets, the `dualtree-kd` algorithm is fastest, and the `dualtree-ct` algorithm is almost as fast. However, for small datasets, the extra overhead of tree construction and tree traversal is not sufficient to provide good speedup. The `elkan` algorithm, because it holds kN bounds, is able to prune away a huge amount of work; however, maintaining all of these bounds becomes prohibitive with large k and the algorithm exhausts all available memory. The same is true of the `blacklist` algorithm: on the largest datasets, with the largest k values, the space required to maintain blacklists for each node in the stack is too much. The `hamerly` and dual-tree algorithms, on the other hand, are the best-behaved with memory usage and do not have any issues with large N or large k; however, the `hamerly` algorithm is very slow on large datasets because it is not able to prune many points at once.

Similar to the observations about the `blacklist` algorithm, the tree-based approaches are less effective in higher dimensions [39]. In our results, though, we do still see competitive speedup in datasets up to a hundred dimensions or so.

Another clear observation is that when k is scaled on a single dataset, the `dualtree-kd` and `dualtree-ct` algorithms nearly always scale better (in terms of runtime) than the other algorithms. These results show that our algorithm satisfies its original goals: to be able to scale effectively to large k and N.

7.8.8 Future directions

Using four pruning strategies, we have developed a flexible, tree-independent dual-tree k-means algorithm that is the best-performing algorithm for large datasets and large k in small-to-medium dimensions. It is theoretically favorable, with the first provably sub-$O(kN)$ bounds (though these depend on dataset-dependent constants), has a small memory footprint, and may be used in conjunction with initial point selection and approximation or sampling schemes to provide additional speedup.

There are still interesting future directions to pursue, though. The first direction is parallelism: because our dual-tree algorithm is agnostic to the type of traversal used, we may use a parallel traversal [28], such as an adapted version of a recent parallel dual-tree algorithm [76]. This parallelism can either be at the single-node level, allowing faster clustering on a single system, or at a larger scale, allowing distributed k-means clustering on a large set of systems. Potentially, this could allow k-means to be an effective strategy for extremely large data which traditionally has been handled with single-pass algorithms.

The second direction is kernel k-means and other spectral clustering techniques: it is possible to merge the dual-tree algorithm here with ideas from the dual-tree algorithm for max-kernel search to perform kernel k-means. This work thus opens promising avenues to further accelerated clustering algorithms. In addition, because spectral clustering has a connection to nonnegative matrix factorization [166], there may be further extensions in that direction: it may be possible to adapt the k-means algorithm given here to the seemingly

completely different problem of matrix factorization for fast collaborative filtering.

CHAPTER 8

CONCLUSION AND FUTURE DIRECTIONS

Throughout this thesis three things have been quite clear:

1. There is no end of large-scale statistical problems, and 'large-scale' is getting larger every day.

2. Hierarchical representations (trees) can provide alleviation to these large-scale problems.

3. Most of these tree-based approaches have some serious similarity.

It is rooted in this perspective that the contributions of this thesis are most apparent and resonant. By unifying the class of dual-tree algorithms in Chapter 3 (and also the class of single-tree algorithms), the path to improvement becomes deobfuscated. The logical split of a dual-tree algorithm into a *type of space tree*, *pruning dual-tree traversal*, and *problem-specific* `BaseCase()` *and* `Score()` *functions* allows consideration of each piece individually, as opposed to intertwined (which was the way of most previous improvements).

Chapter 4 showed how the **mlpack** library can exploit this logical split to provide a clean interface for programmers and an efficient library for users; then, each subsequent chapter detailed improvements for one of the three pieces of dual-tree algorithms: Chapter 5 focused on the improvement of trees—primarily theoretical improvemnts. Chapter 6 detailed an improved dual depth-first traversal that can be applied to any situation. Chapter 7, by far the longest, detailed a cabal of problems that can be solved with dual-tree algorithms, describing each one as a combination of a `BaseCase()` and `Score()` function. Although in many cases empirical results were shown for only one or two types of trees, it cannot be emphasized enough that these `BaseCase()` and `Score()` functions can be

used with *any* combination of tree type and traversal. **mlpack** exploits this fact, providing easily-configurable yet fast algorithms.

The original thesis statement, laid clear in Chapter 1, states that we may improve and expand the class of dual-tree algorithms by focusing on and providing improvements for each of the three independent components of a dual-tree algorithm. All of the evidence in the past horde of pages supports this statement, and therefore I consider my justification of the thesis complete.

But, I am not yet done writing. No work is ever complete, and I think it is important to highlight a few issues in the field of tree-based algorithms that warrant further investigation.

High-dimensional data structures. It is well-known that trees scale poorly to high dimensions. Yet, a large amount of data today exists in high dimensions (thousands or more), and traditional tree-based techniques for search are mostly ineffective. Hashing techniques can often provide decent results in high dimension, but these are still not near the speedups seen for low-dimensional datasets with trees.

Automatic selection of algorithm components. One notable disadvantage of the tree-independent dual-tree algorithm abstraction is that it now burdens the practitioner with the choice of tree, the choice of traversal, and the choice of problem to solve (though a practitioner will generally know what problem they want to solve). Especially with the number of choices available, making an informed choice is something of a daunting task that requires a large amount of experimentation. However, simple heuristics like those employed by Muja and Lowe [206] could provide at least a partial solution to this problem and help to assemble an auto-tuned black box that generally makes decent choices for the parameters. For this to happen, though, a better understanding of the dataset characteristics that make trees more or less effective is required.

More reasonable and tight runtime bounds. The expansion constant is to date the only quantity that has allowed more descriptive runtime bounds for dual-tree algorithms. However, it has some serious drawbacks: it is sensitive to individual points and outliers,

it is extremely time-consuming to calculate[1], and it is not necessarily a great indicator of the performance of trees [57]. Worse yet, the bounds that are shown are with respect to particularly large powers of the expansion constant. These bounds are useful in that they show the scaling properties of the algorithm, but they are not useful in practice, for the reasons listed above. A more robust notion of intrinsic dimensionality could pave the way towards better and more practically usable runtime bounds for dual-tree algorithms.

To my mind, these three issues are the most important issues that need to be addressed in the tree-based algorithm literature, and it is my hope that in the time that follows, I will be able to make attempts at solving these problems.

[1] There is a relatively straightforward $O(N^2 \log N)$ algorithm to do this calculation; but it would be difficult to make it any faster while still being exact, which it would need to be.

REFERENCES

[1] A. Norberg, "High-technology calculation in the early 20th century: Punched card machinery in business and government," *Technology and Culture*, vol. 31, no. 4, pp. 753–779, 1990.

[2] H. Hollerith, "The electrical tabulating machine," *Journal of the Royal Statistical Society*, vol. 57, no. 4, pp. 678–689, 1894.

[3] R. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[4] W. Shockley, *Electrons and Holes In Semiconductors with Applications to Transistor Electronics*. Toronto: D. Van Nostrand Company, Inc., 1950.

[5] J. Von Neumann, "First draft of a report on the EDVAC," tech. rep., Moore School of Electrical Engineering, University of Pennsylvania, June 1945.

[6] F. Hamilton and E. Kubie, "The IBM Magnetic Drum Calculator Type 650," *Journal of the ACM (JACM)*, vol. 1, no. 1, pp. 13–20, 1954.

[7] P. Clark and F. Evans, "Distance to nearest neighbor as a measure of spatial relationships in populations," *Ecology*, vol. 35, no. 4, pp. 445–453, 1954.

[8] G. Cottam and J. Curtis, "The use of distance measures in phytosociological sampling," *Ecology*, vol. 37, no. 3, pp. 451–460, 1956.

[9] M. Dacey, "A note on the derivation of nearest neighbor distances," *Journal of Regional Science*, vol. 2, no. 2, pp. 81–88, 1960.

[10] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[11] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[12] J. Gower and G. Ross, "Minimum spanning trees and single linkage cluster analysis," *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, vol. 18, no. 1, pp. 54–64, 1969.

[13] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[14] A. Dempster, N. Laird, and D. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.

[15] E. Parzen, "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.

[16] C. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[17] G. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," *Numerische Mathematik*, vol. 14, no. 5, pp. 403–420, 1970.

[18] F. Faggin, M. Shima, M. Hoff Jr., H. Feeney, and S. Mazor, "The mcs-4: An lsi micro computer system," in *Proceedings of the 1972 IEEE Region Six Conference*, pp. 8–11, 1972.

[19] "Most important companies," *Byte Magazine*, vol. 20, pp. 99–102, September 1995.

[20] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[21] S. Lohr, "Amid the flood, a catchphrase is born," *The New York Times*, p. BU3, August 12, 2012.

[22] M. Parry, "Please be eAdvised," *The New York Times*, p. ED24, July 22, 2012.

[23] S. Lohr, "The age of Big Data," *The New York Times*, p. SR1, February 12, 2012.

[24] C. Williams and M. Seeger, "Using the Nyström method to speed up kernel machines," in *Advances in Neural Information Processing Systems 13 (NIPS 2000)*, pp. 682–688, 2001.

[25] A. Talwalkar and A. Rostamizadeh, "Matrix coherence and the Nyström method," in *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, pp. 572–579, 2010.

[26] C. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[27] A. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial Intelligence*, vol. 97, no. 1, pp. 245–271, 1997.

[28] R. Curtin, W. March, P. Ram, D. Anderson, A. Gray, and C. Isbell Jr, "Tree-independent dual-tree algorithms," in *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, 2013.

[29] R. Finkel and J. Bentley, "Quad trees: a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[30] C. Jackins and S. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[31] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[32] J. Friedman, J. Bentley, and R. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[33] K. Fukunaga and P. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 750–753, 1975.

[34] W. Koontz, P. Narendra, and K. Fukunaga, "A branch and bound clustering algorithm," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 908–915, 1975.

[35] P. Narendra and K. Fukunaga, "A branch and bound algorithm for feature subset selection," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 917–922, 1977.

[36] J. Bentley and J. Friedman, "Fast algorithms for constructing minimal spanning trees in coordinate spaces," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 97–105, 1978.

[37] J. Bentley and J. Friedman, "Data structures for range searching," *ACM Computing Surveys (CSUR)*, vol. 11, no. 4, pp. 397–409, 1979.

[38] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[39] D. Pelleg and A. Moore, "Accelerating exact *k*-means algorithms with geometric reasoning," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '99)*, pp. 277–281, ACM, 1999.

[40] A. W. Moore, "Very fast em-based mixture model clustering using multiresolution kd-trees," *Advances in Neural Information Processing Systems*, pp. 543–549, 1999.

[41] D. Fussell and K. Subramanian, "Fast ray tracing using kd-trees," Tech. Rep. TR-88-07, The University of Texas at Austin, Department of Computer Sciences, March 1988.

[42] J. Barnes and P. Hut, "A hierarchical *o*(*nlogn*) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[43] Y. Shen, A. Ng, and M. Seeger, "Fast Gaussian process regression using kd-trees," *Advances in Neural Information Processing Systems (NIPS)*, vol. 18, p. 1225, 2006.

[44] K. Deng and A. Moore, "Multiresolution instance-based learning," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI '95)*, vol. 2, pp. 1233–1239, 1995.

[45] P. Ram and A. Gray, "Maximum inner-product search using cone trees," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*, (New York, NY, USA), pp. 931–939, ACM, 2012.

[46] R. Curtin, P. Ram, and A. Gray, "Fast exact max-kernel search," in *SIAM International Conference on Data Mining (SDM '13)*, pp. 1–9, 2013.

[47] A. Guttman, "R-Trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 47–57, ACM, 1984.

[48] S. Omohundro, "Five balltree construction algorithms," Tech. Rep. TR-89-063, University of California, Berkeley International Computer Science Institute Technical Reports, 1989.

[49] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*, pp. 322–331, ACM, 1990.

[50] P. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pp. 311–321, Society for Industrial and Applied Mathematics, 1993.

[51] J. Uhlmann, "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991.

[52] I. Kamel and C. Faloutsos, "Hilbert R-tree: an improved R-tree using fractals," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pp. 500–509, ACM, 1994.

[53] K. Lin, H. Jagadish, and C. Faloutsos, "The TV-tree: an index structure for high-dimensional data," *The VLDB Journal: The International Journal on Very Large Data Bases - Spatial Database Systems*, vol. 3, no. 4, pp. 517–542, 1994.

[54] S. Berchtold, D. Keim, and H.-P. Kriegel, "The X-tree: an index structure for high-dimensional data," in *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*, pp. 28–39, ACM, 1996.

[55] J. McNames, "A fast nearest-neighbor algorithm based on a principal axis search tree," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 9, pp. 964–976, 2001.

[56] T. Liu, A. Moore, K. Yang, and A. Gray, "An investigation of practical approximate nearest neighbor algorithms," in *Advances in Neural Information Processing Systems 18 (NIPS '04)*, pp. 825–832, Neural Information Processing Systems Foundation, 2004.

[57] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, vol. 23, pp. 97–106, 2006.

[58] M. Holmes, A. Gray, and C. Isbell Jr., "QUIC-SVD: Fast SVD using cosine trees," *Advances in Neural Information Processing Systems (NIPS)*, vol. 21, 2008.

[59] P. Ram, D. Lee, and A. Gray, "Nearest-neighbor search on a time budget via max-margin trees," in *SIAM International Conference on Data Mining (SDM '12)*, pp. 1011–1022, 2012.

[60] L. Greengard and V. Rokhlin Jr., "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, no. 2, pp. 325–348, 1987.

[61] A. Gray and A. Moore, "'N-Body' problems in statistical learning," in *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, vol. 4, pp. 521–527, 2001.

[62] S. Aluru, "Greengard's *n*-body algorithm is not order *n*," *SIAM Journal on Scientific Computing*, vol. 17, no. 3, pp. 773–776, 1996.

[63] E. Schein, *DEC is Dead, Long Live DEC*. Berrett-Koehler Publishers, August 2004.

[64] A. Gray, *Bringing Tractability to Generalized N-Body Problems in Statistical and Scientific Computation*. PhD thesis, Carnegie Mellon University, 2003.

[65] A. Gray and A. Moore, "Nonparametric density estimation: Toward computational tractability," in *SIAM International Conference on Data Mining (SDM)*, pp. 203–211, 2003.

[66] R. Curtin, "Faster dual-tree traversal for nearest neighbor search," in *Proceedings of The 8th International Conference on Similarity Search and Applications (SISAP 2015), to appear*, 2015.

[67] P. Ram, D. Lee, H. Ouyang, and A. Gray, "Rank-approximate nearest neighbor search: Retaining meaning and speed in high dimensions," *Advances in Neural Information Processing Systems*, vol. 22, 2009.

[68] W. March, P. Ram, and A. Gray, "Fast Euclidean minimum spanning tree: algorithm, analysis, and applications," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pp. 603–612, 2010.

[69] A. Gray and A. Moore, "Rapid evaluation of multiple density models," in *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics (AISTATS '03)*, 2003.

[70] M. Holmes, A. Gray, and C. Isbell Jr., "Fast nonparametric conditional density estimation," in *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence (UAI '07)*, 2007.

[71] N. Slagle and L. Fortnow, "Approximate matrix multiplication and space partitioning trees: An exploration," tech. rep., Technical report, http://www.npslagle.info, 2012.

[72] P. Wang, D. Lee, A. Gray, and J. Rehg, "Fast mean shift with accurate and stable convergence," in *Workshop on Artificial Intelligence and Statistics (AISTATS)*, 2007.

[73] D. Lee, A. Gray, and A. Moore, "Dual-tree fast Gauss transforms," in *Advances in Neural Information Processing Systems 18*, pp. 747–754, Cambridge, MA: MIT Press, 2006.

[74] D. Lee and A. Gray, "Faster Gaussian summation: Theory and experiment," in *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI '06)*, 2006.

[75] D. Lee and A. Gray, "Fast high-dimensional kernel summations using the monte carlo multipole method," *Advances in Neural Information Processing Systems (NIPS)*, vol. 21, 2008.

[76] D. Lee, R. Vuduc, and A. Gray, "A distributed kernel summation framework for general-dimension machine learning.," in *SIAM International Conference on Data Mining (SDM '12)*, pp. 391–402, 2012.

[77] W. March, A. Connolly, and A. Gray, "Fast algorithms for comprehensive n-point correlation estimates," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*, pp. 1478–1486, ACM, 2012.

[78] W. B. March, K. Czechowski, M. Dukhan, T. Benson, D. Lee, A. J. Connolly, R. Vuduc, E. Chow, and A. G. Gray, "Optimizing the computation of n-point correlations on large-scale astronomical data," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 74, IEEE Computer Society Press, 2012.

[79] R. Curtin and P. Ram, "Dual-tree fast exact max-kernel search," *Statistical Analysis and Data Mining*, vol. 7, no. 4, pp. 229–253, 2014.

[80] R. Curtin, "Dual-tree k-means clustering with bounded single-iteration runtime," in *submitted to Neural Information Processing Systems 28 (NIPS 2015)*, 2015.

[81] M. Klaas, M. Briers, N. de Freitas, A. Doucet, S. Maskell, and D. Lang, "Fast particle smoothing: if i had a million particles," in *Proceedings of the 23rd International Conference on Machine learning (ICML '06)*, pp. 481–488, ACM, 2006.

[82] L. Van Der Maaten, "Accelerating T-SNE using tree-based algorithms," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3221–3245, 2014.

[83] A. Gray, "Fast kernel matrix-vector multiplication with application to Gaussian process learning," Tech. Rep. CMU-CS-04-1103, School of Computer Science, Carnegie Mellon University, 2004.

[84] B. Thiesson and J. Kim, "Fast variational mode-seeking," in *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2012)*, pp. 1230–1242, 2012.

[85] S. Amizadeh, B. Thiesson, and M. Hauskrecht, "Variational dual-tree framework for large-scale transition matrix approximation," in *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence (UAI '12)*, pp. 64–73, 2012.

[86] C. Puech and H. Yahia, "Quadtrees, octrees, hyperoctrees: a unified analytical approach to tree data structures used in graphics, geometric modeling and image processing," in *Proceedings of The First Annual Symposium on Computational Geometry (SoCG '85)*, pp. 272–280, ACM, 1985.

[87] R. Curtin, J. Cline, N. Slagle, W. March, P. Ram, N. Mehta, and A. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[88] M. Shamos and D. Hoey, "Closest-point problems," in *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS 1975)*, pp. 151–162, IEEE, 1975.

[89] N. Megiddo, E. Zemel, and S. Hakimi, "The maximum coverage location problem," *SIAM Journal on Algebraic Discrete Methods*, vol. 4, no. 2, pp. 253–261, 1983.

[90] J. Ritter, "An efficient bounding sphere," in *Graphics Gems*, pp. 301–303, Academic Press Professional, Inc., 1990.

[91] M. Bādoiu, S. Har-Peled, and P. Indyk, "Approximate clustering via core-sets," in *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC 2002)*, pp. 250–257, ACM, 2002.

[92] P. Kumar, J. Mitchell, and E. Yildirim, "Approximate minimum enclosing balls in high dimensions using core-sets," *Journal of Experimental Algorithmics*, vol. 8, pp. 1–29, 2003.

[93] K. Fischer, B. Gärtner, and M. Kutz, "Fast smallest-enclosing-ball computation in high dimensions," in *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, pp. 630–641, Springer, 2003.

[94] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[95] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, vol. 10, pp. 10–17, 2010.

[96] The Apache Foundation, "Apache Mahout," June 2015. Project homepage and software download at `http://mahout.apache.org/`.

[97] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and D. E., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[98] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello, "mlpy: Machine Learning PYThon," 2012. Project homepage at `http://mlpy.fbk.eu/`.

[99] A. Ben-Hur and J. Weston, "PyML-machine learning in Python." `http://pyml.sourceforge.net`, 2015.

[100] K. Gawande, C. Webers, A. Smola, and S. Vishwanathan, "ELEFANT: A python machine learning toolbox," in *SciPy Conference*, 2007.

[101] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "Pybrain," *The Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.

[102] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio, "Theano: Deep learning on GPUs with python," in *Proceedings of the NIPS 2011 BigLearning Workshop*, 2011.

[103] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations*, vol. 11, no. 1, 2009.

[104] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[105] D. Eddelbuettel, R. François, J. Allaire, J. Chambers, D. Bates, and K. Ushey, "Rcpp: Seamless r and c++ integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011.

[106] J. Langford, L. Li, and A. Strehl, "Vowpal Wabbit online learning project," tech. rep., http://hunch.net, 2007.

[107] S. Sonnenburg, G. Rätsch, S. Henschel, C. Widmer, J. Behr, A. Zien, F. d. Bona, A. Binder, C. Gehl, and V. Franc, "The SHOGUN machine learning toolbox," *The Journal of Machine Learning Research*, vol. 11, pp. 1799–1802, 2010.

[108] C. Igel, V. Heidrich-Meisner, and T. Glasmachers, "Shark," *The Journal of Machine Learning Research*, vol. 9, pp. 993–996, 2008.

[109] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Indianapolis: Addison-Wesley Professional, Feb. 2001.

[110] C. Sanderson, "Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments," tech. rep., NICTA, 2010.

[111] B. Schölkopf, A. Smola, and K. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural Computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[112] S. Burer and R. Monteiro, "A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization," *Mathematical Programming*, vol. 95, no. 2, pp. 329–357, 2003.

[113] M. Edel, A. Soni, and R. Curtin, "An automatic benchmarking system," in *Proceedings of the NIPS 2014 Workshop on Software Engineering for Machine Learning (SE4ML)*, 2014.

[114] R. Schapire and Y. Singer, "Improved boosting algorithms using confidence-rated predictions," *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999.

[115] D. Lee and H. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788–791, 1999.

[116] C.-C. Ma, "A guide to singular value decomposition for collaborative filtering," 2008.

[117] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *The Annals of statistics*, vol. 32, no. 2, pp. 407–499, 2004.

[118] P. Ram and A. Gray, "Density estimation trees," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 627–635, ACM, 2011.

[119] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.

[120] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SoCG '04)*, pp. 253–262, ACM, 2004.

[121] J. Goldberger, G. Hinton, S. Roweis, and R. Salakhutdinov, "Neighbourhood components analysis," in *Advances in Neural Information Processing Systems 17 (NIPS 2004)*, pp. 513–520, 2004.

[122] K. Yu, T. Zhang, and Y. Gong, "Nonlinear learning using Local Coordinate Coding," in *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, pp. 2223–2231, 2009.

[123] H. Lee, A. Battle, R. Raina, and A. Ng, "Efficient sparse coding algorithms," in *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pp. 801–808, 2006.

[124] S. Agrawal, R. Curtin, S. Ghaisas, and M. Gupta, "Collaborative filtering via matrix decomposition in mlpack," in *Proceedings of the ICML 2015 Workshop on Machine Learning Open Source Software*, 2015.

[125] A. Moore, "The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data," in *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI '00)*, pp. 397–405, Morgan Kaufmann Publishers Inc., 2000.

[126] J. McClellan, R. Schafer, and M. Yoder, *Signal Processing First*. Prentice Hall, 2003.

[127] N. Segata and E. Blanzieri, "Fast and scalable local kernel machines," *The Journal of Machine Learning Research*, vol. 99, pp. 1883–1926, 2010.

[128] B. Liu and H. Jagadish, "Using trees to depict a forest," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 133–144, 2009.

[129] A. Kibriya and E. Frank, "An empirical comparison of exact nearest neighbour algorithms," in *Knowledge Discovery in Databases: PKDD 2007*, pp. 140–151, Springer, 2007.

[130] C. Parker, A. Fern, and P. Tadepalli, "Learning for efficient retrieval of structured data with noisy queries," in *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, pp. 729–736, ACM, 2007.

[131] D. Karger and M. Ruhl, "Finding nearest neighbors in growth-restricted metrics," in *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC '02)*, pp. 741–750, ACM, 2002.

[132] R. Krauthgamer and J. Lee, "Navigating nets: simple algorithms for proximity search," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 798–807, Society for Industrial and Applied Mathematics, 2004.

[133] P. Ram, D. Lee, W. March, and A. Gray, "Linear-time algorithms for pairwise statistical problems," *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, vol. 23, 2009.

[134] K. Bache and M. Lichman, "UCI Machine Learning Repository," 2013. `http://archive.ics.uci.edu/ml`.

[135] R. Curtin, D. Lee, W. March, and P. Ram, "Plug-and-play dual-tree algorithm runtime analysis," *Journal of Machine Learning Research (to appear)*, 2015.

[136] D. Colless, "Review of 'Phylogenetics: The Theory and Practice of Phylogenetic Systematics', by E.O. Wiley," *Systematic Zoology*, vol. 31, pp. 100–104, 1982.

[137] M. Sackin, ""Good" and "bad" phenograms," *Systematic Biology*, vol. 21, no. 2, pp. 225–226, 1972.

[138] Y. LeCun, C. Cortes, and C. Burges, "MNIST dataset," 2000. http://yann.lecun.com/exdb/mnist/.

[139] J. Adelman-McCarthy, M. Agüeros, S. Allam, C. Prieto, K. Anderson, S. Anderson, J. Annis, N. Bahcall, C. Bailer-Jones, I. Baldry, *et al.*, "The sixth data release of the Sloan Digital Sky Survey," *The Astrophysical Journal Supplement Series*, vol. 175, no. 2, p. 297, 2008.

[140] D. Moore and S. Russell, "Fast Gaussian process posteriors with product trees," in *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI-14)*, (Quebec City), July 2014.

[141] M. Houle and M. Nett, "Rank cover trees for nearest neighbor search," in *Proceedings of the 6th International Conference on Similarity Search and Applications (SISAP 2013)*, pp. 16–29, Springer, 2013.

[142] P. Li, M. Wang, J. Cheng, C. Xu, and H. Lu, "Spectral hashing with semantically consistent graph for image indexing," *IEEE Transactions on Multimedia*, vol. 15, no. 1, pp. 141–152, 2013.

[143] N. Tziortziotis, C. Dimitrakakis, and K. Blekas, "Cover tree bayesian reinforcement learning," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 2313–2335, 2014.

[144] A. Kibriya, "Fast algorithms for nearest neighbor search," Master's thesis, University of Waikato, 2007.

[145] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: A new data clustering algorithm and its applications," *Data Mining and Knowledge Discovery*, vol. 1, no. 2, pp. 141–182, 1997.

[146] R. Lupton, J. Gunn, Z. Ivezic, G. Knapp, and S. Kent, "The SDSS imaging pipelines," in *Astronomical Data Analysis Software and Systems X*, vol. 238, p. 269, 2001.

[147] J. Adelman-McCarthy, M. Agüeros, S. Allam, C. Prieto, K. Anderson, *et al.*, "The sixth data release of the Sloan Digital Sky Survey," *The Astrophysical Journal Supplement Series*, vol. 175, no. 2, p. 297, 2008.

[148] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling LSH for performance tuning," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM 2008)*, pp. 669–678, ACM, 2008.

[149] W. Dong. Personal communication, 2015.

[150] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[151] J. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, pp. 599–608, ACM, 1997.

[152] S. Nene and S. Nayar, "A simple algorithm for nearest neighbor search in high dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 9, pp. 989–1003, 1997.

[153] P. Yianilos, "Excluded middle vantage point forests for nearest neighbor search," in *In DIMACS Implementation Challenge, ALENEX'99*, 1999.

[154] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pp. 604–613, ACM, 1998.

[155] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases (VLDB '99)*, vol. 99, pp. 518–529, 1999.

[156] K. Clarkson, "Nearest neighbor queries in metric spaces," *Discrete & Computational Geometry*, vol. 22, no. 1, pp. 63–93, 1999.

[157] K. Clarkson, "Nearest-neighbor searching and metric space dimensions," *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pp. 15–59, 2006.

[158] K. Clarkson, "Fast algorithms for the all nearest neighbors problem," in *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, pp. 226–232, 1983.

[159] R. Curtin, J. Cline, N. Slagle, M. Amidon, and A. Gray, "MLPACK: A Scalable C++ Machine Learning Library," in *BigLearning: Algorithms, Systems, and Tools for Learning at Scale*, 2011.

[160] P. Agarwal and J. Erickson, "Geometric range searching and its relatives," *Contemporary Mathematics*, vol. 223, pp. 1–56, 1999.

[161] J. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[162] A. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, 2004.

[163] T. Jaakkola and D. Haussler, "Probabilistic kernel regression models," in *Proceedings of the Seventh International Workshop on Artificial Intelligence and Statistics (AISTATS '99)*, 1999.

[164] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[165] I. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: spectral clustering and normalized cuts," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*, pp. 551–556, ACM, 2004.

[166] C. Ding, X. He, and H. Simon, "On the equivalence of Nonnegative Matrix Factorization and spectral clustering," in *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM '05)*, pp. 606–610, SIAM, 2005.

[167] C. Rasmussen, "Gaussian Processes for machine learning," 2006.

[168] C. Williams, "Prediction with Gaussian Processes: from linear regression to linear prediction and beyond," in *Learning in Graphical Models*, pp. 599–621, Springer, 1998.

[169] B. Schölkopf, A. Smola, and K. Müller, "Kernel principal component analysis," in *Advances In Kernel Methods: Support Vector Learning*, pp. 327–352, MIT Press, 1999.

[170] S. Günter, N. Schraudolph, and S. Vishwanathan, "Fast iterative kernel principal component analysis," *Journal of Machine Learning Research*, vol. 8, pp. 1893–1918, 2007.

[171] L. Amaro, Z. Heiga, Y. Nankaku, C. Miyajima, K. Tokuda, and T. Kitamura, "On the use of kernel PCA for feature extraction in speech recognition," *IEICE Transactions on Information and Systems*, vol. 87, no. 12, pp. 2802–2811, 2004.

[172] T. Takiguchi and Y. Ariki, "Robust feature extraction using kernel PCA," in *Proceedings of the 2006 IEEE International Conference on Speech and Signal Processing (ICASSP 2006)*, 2006.

[173] C. Liu, "Gabor-based kernel PCA with fractional power polynomial models for face recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 5, pp. 572–581, 2004.

[174] H. Hoffman, "Kernel PCA for novelty detection," *Pattern Recognition*, vol. 40, no. 3, pp. 863–874, 2007.

[175] Y. Rathi, S. Dambreville, and A. Tannenbaum, "Statistical shape analysis using kernel PCA," in *Proceedings of the SPIE 6064, Image Processing: Algorithms and Systems, Neural Networks, and Machine Learning*, 2006.

[176] M. López, J. Ramírez, J. Górriz, I. Álvarez, D. Salas-Gonzalez, F. Segovia, and R. Chaves, "SVM-based CAD system for early detection of Alzheimer's disease using kernel PCA and LDA," *Neuroscience Letters*, vol. 464, no. 3, pp. 233–238, 2009.

[177] M. Genton, "Classes of kernels for machine learning: a statistics perspective," *The Journal of Machine Learning Research*, vol. 2, pp. 299–312, 2002.

[178] J. Ham, D. Lee, S. Mika, and B. Schölkopf, "A kernel view of the dimensionality reduction of manifolds," in *Proceedings of the 21st International Conference on Machine Learning (ICML '04)*, pp. 47–54, 2004.

[179] B. Hamers, J. Suykens, and B. De Moor, "Compactly supported RBF kernels for sparsifying the Gram matrix in LS-SVM regression models," in *Artificial Neural Networks – ICANN 2002*, pp. 720–726, Springer, 2002.

[180] I. Murray, "Gaussian processes and fast matrix-vector multiplies," 2009. In the Numerical Mathematics in Machine Learning Workshop at ICML '09.

[181] H. Zhang, M. Genton, and P. Liu, "Compactly supported radial basis function kernels," Tech. Rep. 2570, North Carolina State University Department of Statistics Technical Reports, 2004.

[182] S. Mika, B. Schölkopf, A. Smola, K. Müller, M. Scholz, and G. Rätsch, "Kernel pca and de-noising in feature spaces.," in *Advances in Neural Information Processing Systems 11 (NIPS '97)*, pp. 536–542, 1998.

[183] V. Franc and V. Hlavác, "Statistical pattern recognition toolbox for Matlab," no. CTU-CMP-2004-08, 2004.

[184] A. Smola and B. Schölkopf, "Sparse greedy matrix approximation for machine learning," in *Proceedings of the 17th International Conference on Machine Learning (ICML '00)*, pp. 911–918, 2000.

[185] A. Frieze, R. Kannan, and S. Vempala, "Fast monte-carlo algorithms for finding low-rank approximations," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 1025–1041, 2004.

[186] A. Gittens, "The spectral norm error of the naive nystrom extension," *arXiv preprint arXiv:1110.5305*, 2011.

[187] R. Lehoucq, D. Sorensen, and C. Yang, *ARPACK User's Guide: Solution of Large-Scale Eigenvalue Problems With Implicityly Restorted Arnoldi Methods*, vol. 6. SIAM, 1998.

[188] B. Schölkopf, "The kernel trick for distances," *Advances in Neural Information Processing Systems 13 (NIPS '01)*, pp. 301–307, 2001.

[189] K. Zhang, I. Tsang, and J. Kwok, "Improved Nyström low-rank approximation and error analysis," in *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, pp. 1232–1239, ACM, 2008.

[190] R. Weber, H. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proceedings of the 24th International Conference on Very Large Databases (VLDB '98)*, pp. 194–205, 1998.

[191] S. Mika, G. Rätsch, J. Weston, B. Schölkopf, and K. Müller, "Fisher discriminant analysis with kernels," in *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.*, pp. 41–48, IEEE, 1999.

[192] F. Bach and M. Jordan, "Kernel independent component analysis," *The Journal of Machine Learning Research*, vol. 3, pp. 1–48, 2003.

[193] S. Si, C.-J. Hsieh, and I. Dhillon, "Memory efficient kernel approximation," in *Proceedings of the Thirty-First International Conference on Machine Learning (ICML '14)*, pp. 701–709, 2014.

[194] W. March, B. Xiao, S. Tharakan, D. Chenhan, and G. Biros, "Robust treecode approximation for kernel machines," in *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*, 2015.

[195] R. Curtin, "Single-tree GMM training," Tech. Rep. GT-CSE-2015-01, Georgia Institute of Technology, School of Computational Science and Engineering, 2015.

[196] D. Reynolds, "Gaussian mixture models," in *Encyclopedia of Biometrics*, pp. 659–663, 2009.

[197] J. Bilmes, "A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models," Tech. Rep. TR-97-021, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1998.

[198] C. Leslie, E. Eskin, and W. Noble, "The spectrum kernel: A string kernel for SVM protein classification," in *Proceedings of the Pacific Symposium on Biocomputing*, pp. 564–575, 2002.

[199] K. Borgwardt, C. Ong, S. Schönauer, S. V. N. Vishwanathan, A. Smola, and H. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, no. suppl. 1, pp. i47–i56, 2005.

[200] K. Müller, A. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik, "Predicting time series with Support Vector Machines," *Proceedings of the 7th International Conference on Artificial Neural Networks (ICANN '97)*, pp. 999–1004, 1997.

[201] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[202] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11," *Journal of Machine Learning Research (Proceedings Track)*, vol. 18, pp. 8–18, 2012.

[203] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[204] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[205] V. Hodge and J. Austin, "A comparison of standard spell checking algorithms and a novel binary neural approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1073–1081, 2003.

[206] M. Muja and D. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Applications (VISAPP)*, 2009.

[207] A. Rahimi and B. Recht, "Random Features for Large-scale Kernel Machines," *Advances in Neural Information Processing Systems 20 (NIPS '07)*, pp. 1177–1184, 2008.

[208] P. Kar and H. Karnick, "Random feature maps for dot product kernels," in *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS '12)*, vol. 22, pp. 583–591, 2012.

[209] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pp. 380–388, 2002.

[210] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Proceedings of the 12th IEEE International Conference on Computer Vision (ICCV '09)*, 2009.

[211] L. Cayton, "Fast nearest neighbor retrieval for Bregman divergences," in *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, pp. 112–119, 2008.

[212] S. Dasgupta and Y. Freund, "Random projection trees and low dimensional manifolds," in *Proceedings of the 40th Annual ACM Symposium on Theory Of Computing (STOC '08)*, (New York, NY, USA), pp. 537–546, ACM, 2008.

[213] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer, 1985.

[214] J. Bennett and S. Lanning, "The Netflix Prize," in *Proceedings of the KDD Cup and Workshop*, pp. 3–6, 2007.

[215] S. Kim, F. Li, G. Lebanon, and I. Essa, "Beyond sentiment: The manifold of human emotions," in *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS '13)*, pp. 360–369, 2013.

[216] A. Torralba, R. Fergus, and W. Freeman, "80 Million Tiny Images: A large data set for nonparametric object and scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1958–1970, 2008.

[217] W. Pearson and D. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences*, vol. 85, no. 8, pp. 2444–2448, 1988.

[218] P. Bradley and U. Fayyad, "Refining initial points for k-means clustering.," in *Proceedings of the 15th International Conference on Machine Learning (ICML '98)*, pp. 91–99, 1998.

[219] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035, 2007.

[220] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proceedings of the 20th International Conference on Machine Learning (ICML '03)*, vol. 3, pp. 147–153, 2003.

[221] G. Hamerly, "Making *k*-means even faster," in *Proceedings of the 2010 SIAM International Conference on Data Mining*, pp. 130–140, 2010.

[222] G. Frahling and C. Sohler, "A fast *k*-means implementation using coresets," *International Journal of Computational Geometry & Applications*, vol. 18, no. 06, pp. 605–625, 2008.

[223] Y. Kwon, D. Nunley, J. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*, pp. 132–150, Springer, 2010.

[224] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on Statistical Learning in Computer Vision, ECCV*, vol. 1, pp. 1–16, 2004.

[225] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of AISTATS*, pp. 215–223, 2011.

[226] S. Bengio, J. Weston, and D. Grangier, "Label embedding trees for large multi-class tasks," in *Advances in Neural Information Processing Systems 23 (NIPS '10)*, vol. 23, p. 3, 2010.

[227] F. Can and E. Ozkarahan, "Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases," *ACM Transactions on Database Systems*, vol. 15, pp. 483–517, Dec. 1990.

[228] M. Izbicki and C. Shelton, "Faster cover trees," in *Proceedings of the Thirty-Second International Conference on Machine Learning (ICML '15)*, 2015.

[229] D. Pelleg and A. Moore, "X-means: extending *k*-means with efficient estimation of the number of clusters," in *Proceedings of the 17th International Conference on Machine Learning (ICML '00)*, pp. 727–734, 2000.